See MIPS Run Linux (2nd edition)

看 MIPS 跑 Linux （第二版）

———MIPS 体系结构剖析

D. Sweetman（斯维特曼）著
J.Qu（屈建勤）译

2007 年 12 月 31 日

译者说明

这里的译文原本出于教学的需要而进行的，因而在翻译时力求准确，并尽量保持原书的语言风格。在不损害原作者和出版社的知识产权的情况下，译者允许和鼓励任何人出于教育、学习或科学研究的非盈利目的，复制和传播本文档的电子拷贝。该书英文原版由 Elsevier 出版，版权归原作者和出版社所有。有关英文原版的任何问题，请联系原作者和原出版社。

以下说说译者在翻译时的一些考虑。

译文力求贴近原文。原书的文字比较生动，风格比较明显。翻译时首先尽量保证译文的准确性。在此基础之上，尽量贴近原文的风格。原文中涉及到典故、成语或其它文化背景的地方，尽量翻译成中文中类似的说法。比如第 4 章关于“skimmed the cream”和第 13 章标题的翻译。

对于原书中的插图尽量照原样保留，其中的文字不作翻译。这是考虑到插图本身就比较直观，插图中的少量英文图例不应当对读者阅读造成障碍。

对于原书后面的索引，在译文中直接略去。这样做是考虑到这种索引对于电子文档的意义不大。同时索引中所涉及的英文在正文多数已经被翻译，原本按照字母次序组织的索引该如何重新组织也要考虑，但索引本身的意义已经不大，似乎不值得为此费神。

对于本书的书名的翻译，翻译时做了特别考虑。该书的第一版中文将书名译成《MIPS 处理器设计透视》，其错误实在是过于离谱。一本介绍体系结构的面向程序员的书，愣是把书名给改成让人误以为是一本介绍处理器设计的面向硬件人员的书，完全违背了原书的本意，且有误导读者之嫌。本来，秉承原书的风格，本书的书名直译为《看 MIPS 跑 Linux》就很形象。但是这种过于形象化的名称或许不太符合中文读者对科技类书名的期望，因此翻译时给书名加上了一个反映书的内容的副标题。

对于原书中的一些失误，翻译时根据情况作了不同的处理。比如原书的图 2.2，经过向原作者反映确认后进行了修正。而一些明显的笔误或者排版错误，则直接予以修正，而不一一注明。另一些难以简单修正的失误，则照原文译出。

对于一些专业词汇，尽量采用通用的规范的译法。有些专业词汇有两种以上较通用的译法（比如 write-through 译为“透写”和“通写”），在译文中选用一种，并且保持前后一致。在缺乏公认的规范译法的情况下，结合上下文的含义以意

译为主。个别情况下参考某些译法进行修订，或者根据译者的理解直接翻译。例如对于 cache alias 的翻译，以及第七章对于两种非数 NaN 的翻译等等。

第十二章对于 JTAG probe 的翻译，处理有所不同。一种较为广泛的译法是译成“探针，” 译者也采用了这种译法。但“探针”一词在缺乏上下文的情况下容易引起歧义，也容易让不熟悉的人认为它是一种细长的针形设备。有鉴于此，文中也给出了另一种译法，即“接探器。”“接探”一词源于“JTAG”的音译“接探格”——又可以理解为意译，即“一种集成电路的连接测试、探测的标准规格。”这样理解，probe 作为符合“接探格”的设备，译成“接探器”就很自然，兼顾了音和意，也消除了歧义。译文中同时采用了两种译法，但基本保持一致。哪种译法更好，读者可自己判断。

从 2007 年 7 月 27 日的草稿，到 2007 年 9 月 7 日的初稿，其间作了较大的修订。此后又有两次局部的修订，更正了译文中的一些错误。这次作了更全面的修订，修订范围主要是针对书的后半部分。本次修订的幅度较大，除了纠正笔误外，对一些译得不够准确、比较生硬、或者衔接不好的地方也作了不少调整，某些不够清晰的插图重新进行了制作，个别表格排版有轻微变动。此外，这次修订由译者在书后增加了几条与 PIC 代码有关的汇编指示语句的说明，希望对某些读者有所帮助。

尽管译者作了种种努力以保持原文的知识性和趣味性，但译文中存在各种错误疏漏在所难免。如果译文尚有可取之处的话，毫无疑问，这一切首先应当归功于原作者。如果译文表达不够准确或者失去了原文的风采的话，那么首先应当是译者的责任。译文中的种种问题，热情欢迎读者提出宝贵意见。您的批评、意见和建议，请直接发送给译者（jqu@softprise.com）。

译者

2007 年 12 月 31 日

目录

译者说明 **i**

序 **xiii**

前言 **xv**

第 1 章 MIPS 和 RISC 体系结构 **1**

1.1 流水线 2

1.1.1 什么使得流水线效率低下了？ 3

1.1.2 流水线和高速缓存 4

1.2 MIPS 的五级流水线 4

1.3 RISC 和 CISC 6

1.4 过去和现在的一些重要的 MIPS 芯片 7

1.4.1 R2000 到 R3000 处理器 7

1.4.2 R6000 处理器：一度转向 8

1.4.3 第一个 CPU 核 9

1.4.4 QED: 嵌入式系统里的快速 MIPS 处理器 11

1.4.5 R10000 处理器及其后继者 12

1.4.6 消费类电子产品领域的 MIPS 处理器 13

1.4.7 网络路由器和激光打印机领域的 MIPS 13

1.4.8 当今时代的 MIPS 处理器 15

1.4.9 今天 18

1.5 MIPS 和 CISC 体系结构比较 20

1.5.1 MIPS 指令集的限制 20

1.5.2 寻址和访存 21

1.5.3 MIPS 没有的特性 22

1.5.4 程序员可见的流水线效果 23

第 2 章 MIPS 体系结构 **25**

2.1 MIPS 汇编语言的风格初探 28

2.2 寄存器 29
2.2.1 通用寄存器的习惯命名和用法 30
2.3 整数乘法部件及寄存器 32
2.4 加载与存储：寻址方式 33
2.5 存储器与寄存器的数据类型 34
2.5.1 整数数据类型 34
2.5.2 未对齐的加载和存储 35
2.5.3 内存中的浮点数据 35
2.6 汇编语言的合成指令 36
2.7 MIPS I 到 MIPS64 ISA: 64 位(和其它)的扩展 37
2.7.1 发展到 64 位 38
2.7.2 谁需要 64 位？ 39
2.7.3 关于 64 位与 CPU 模式切换：寄存器中的数据 39
2.8 基本地址空间 41
2.8.1 简单系统的寻址 42
2.8.2 核心与用户特权级 43
2.8.3 完整的图像：64 位的地址映射 43
2.9 流水线可见性 44

第 3 章 协处理器 0：MIPS 处理器控制 46
3.1 CPU 控制指令 49
3.2 什么时候要用到哪些寄存器？ 51
3.3 CPU 控制寄存器及其编码 52
3.3.1 状态寄存器（SR） 52
3.3.2 原因寄存器(Cause) 56
3.3.3 异常返回地址(EPC)寄存器 57
3.3.4 无效虚拟地址(BadVaddr)寄存器 57
3.3.5 Count/Compare 寄存器：CPU 片上定时器 57
3.3.6 处理器 ID(PRId) 寄存器 59
3.3.7 Config 寄存器：CPU 资源信息和配置 60
3.3.8 EBase 和 IntCtl：中断和异常设置 63
3.3.9 SRSCtl 和 SRSMap：影子寄存器的设置 64
3.3.10 连锁加载地址（LLAddr）寄存器 65
3.4 CP0 遇险——提防落入陷阱 66
3.4.1 遇险防护指令 66
3.4.2 指令遇险和用户遇险 67
3.4.3 在 CP0 指令之间的防护 67

第 4 章 MIPS 的高速缓存工作机制 69
4.1 高速缓存及其管理 69
4.2 高速缓存是怎样工作的 70
4.3 早期 MIPS CPU 的透写高速缓存 72
4.4 MIPS CPU 的回写高速缓存 73
4.5 高速缓存设计的一些其它考虑 73
4.6 管理高速缓存 75
4.7 二级（L2）和三级（L3）高速缓存 77
4.8 MIPS CPU 的高速缓存配置 77
4.9 对 MIPS32/64 高速缓存的编程 78
4.9.1 cache 指令 79
4.9.2 高速缓存初始化和 Tag/Data 寄存器 80
4.9.3 CacheERR、ERR 和 ErrorEPC 寄存器：存储器/高速缓存出错处理 82
4.9.4 确定高速缓存的大小和配置 83
4.9.5 初始化程序 84
4.9.6 作废或者回写高速缓存中的存储区 85
4.10 高速缓存的效率 86
4.11 重新组织软件以影响高速缓存的效率 88
4.12 高速缓存重影 90

第 5 章 异常、中断及初始化 92
5.1 精确异常 93
5.1.1 非精确异常——历史上的 MIPS CPU 中的乘法器 94
5.2 异常发生的时机 95
5.3 异常向量：异常处理开始的地方 95
5.4 异常处理：基本过程 99
5.5 从异常返回 99
5.6 嵌套异常 100
5.7 异常处理例程 100
5.8 中断 101
5.8.1 MIPS CPU 的中断资源 101
5.8.2 在软件中实现中断优先级 103
5.8.3 原子性以及对 SR 的原子修改 104
5.8.4 允许中断的临界区：MIPS 式的信号量 106
5.8.5 MIPS32/64 CPU 中的向量化和 EIC 中断 107
5.8.6 影子寄存器 108
5.9 启动 108
5.9.1 检测和识别你的 CPU 110

5.9.2 引导步骤 111
5.9.3 启动应用程序 111
5.10 指令仿真 112

第 6 章 底层内存管理与 TLB 114
6.1 TLB/MMU 硬件及其作用 114
6.2 TLB/MMU 寄存器 115
6.2.1 TLB 关键字域——EntryHi 和 PageMask 115
6.2.2 TLB 输出域——EntryLo0–1 118
6.2.3 选择 TLB 的表项——**Index**、**Random** 和 **Wired** 寄存器 120
6.2.4 页表存取辅助寄存器—— Context 和 XContext 120
6.3 TLB/MMU 的控制指令 122
6.4 TLB 编程 123
6.4.1 重填是怎样发生的? 123
6.4.2 使用 ASID 124
6.4.3 Random 寄存器和 Wired 寄存器 124
6.5 对硬件友好的页表和重填机制 125
6.5.1 TLB 未命中处理 126
6.5.2 XTLB 未命中处理程序 127
6.6 MIPS TLB 的日常使用 128
6.7 更简单的操作系统中的内存管理 129

第 7 章 浮点支持 131
7.1 有关浮点数的基本概念 131
7.2 IEEE 754 标准及其历史背景 132
7.3 IEEE 浮点数的存储方式 133
7.3.1 IEEE 尾数和规格化 134
7.3.2 为特殊值使用而保留的指数值 135
7.3.3 MIPS FP 数据格式 135
7.4 MIPS 对 IEEE 754 的实现 136
7.4.1 所有 MIPS CPU 都需要浮点自陷处理程序和仿真程序 . . 138
7.5 浮点寄存器 138
7.5.1 浮点寄存器的传统习惯命名和用法 139
7.6 浮点异常/中断 139
7.7 浮点控制: 控制/状态寄存器 140
7.8 浮点实现寄存器 143
7.9 浮点指令指南 144
7.9.1 Load/Store 145
7.9.2 寄存器间传送 146

7.9.3 三操作数算术运算 147
7.9.4 乘加操作 147
7.9.5 单目（变号）操作 148
7.9.6 数据转换操作 148
7.9.7 条件分支和测试指令 149
7.10 单精度对浮点指令以及 MIPS-3D ASE 150
7.10.1 异常和单精度对浮点指令 151
7.10.2 单精度对三操作数算术运算、乘加、变号和无条件数据传送操作 151
7.10.3 单精度对的类型转换操作 152
7.10.4 单精度对的测试和条件传送指令 152
7.10.5 MIPS-3D 指令 153
7.11 指令时序要求 155
7.12 指令时序和速度 156
7.13 即需初始化和使能 156
7.14 浮点仿真 157

第 8 章 MIPS 指令集参考大全 159
8.1 一个简单的例子 159
8.2 汇编指令及其意义 160
8.2.1 带 U 和不带 U (Non-U) 的助记符 162
8.2.2 除法助记符 162
8.2.3 指令清单列表 163
8.3 浮点指令 186
8.4 与 MIPS32/64 第一版的差别 191
8.4.1 第二版增加的普通指令 191
8.4.2 第二版增加的特权指令 193
8.5 特殊指令及其用途 194
8.5.1 向左加载/向右加载：未对齐的加载和存储 194
8.5.2 连锁加载/条件存储 197
8.5.3 条件传送指令 199
8.5.4 可能分支 200
8.5.5 整数乘加和累加指令 200
8.5.6 浮点乘加指令 201
8.5.7 多个浮点条件位 201
8.5.8 预取 202
8.5.9 Sync：用于 load/store 的存储器防护 202
8.5.10 遇险防护指令 204
8.5.11 synci: 为改写指令的程序做高速缓存管理 205

8.5.12 读取硬件寄存器 205
8.6 指令编码 206
8.6.1 指令编码表中的各个域 206
8.6.2 指令编码表的几点注释 221
8.6.3 编码和简单实现 221
8.7 指令按功能分类 221
8.7.1 空操作 222
8.7.2 寄存器/寄存器传送 222
8.7.3 常数加载 222
8.7.4 算术/逻辑运算 223
8.7.5 整数乘法、除法和求余数 224
8.7.6 整数乘（累）加 225
8.7.7 加载和存储 226
8.7.8 跳转、子程序调用和分支 227
8.7.9 断点和自陷 228
8.7.10 CP0 功能：CPU 控制指令 229
8.7.11 浮点指令 229
8.7.12 用户态下对“地下”特性的有限访问 229

第 9 章 阅读 MIPS 汇编语言代码 231
9.1 一个简单的例子 232
9.2 语法概要 236
9.2.1 布局、定界符和标识符 236
9.3 指令的一般规则 237
9.3.1 计算指令：三、二、一个寄存器 237
9.3.2 带立即数的运算指令 237
9.3.3 关于 32/64 位指令 238
9.4 寻址模式 238
9.4.1 相对于 GP 的寻址 240
9.5 目标文件及其在存储器映像中的布局 241
9.5.1 包括堆和栈在内的实际的程序布局 243

第 10 章 向 MIPS 体系结构移植软件 245
10.1 MIPS 应用程序的底层软件：常见问题一览表 245
10.2 尾端：字、字节和位序 246
10.2.1 比特、字节、字和整数 247
10.2.2 软件和尾端问题 249
10.2.3 硬件和尾端 251
10.2.4 MIPS CPU 的双尾端软件 256

10.2.5 可移植性和尾端无关的代码 258
10.2.6 尾端和外来数据 259
10.3 高速缓存可见性带来的问题 259
10.3.1 高速缓存管理和 DMA 数据 261
10.3.2 高速缓存管理和向内存写入指令：自修改代码 262
10.3.3 高速缓存管理和非高速缓存或透写的数据 263
10.3.4 高速缓存重影和页面着色 263
10.4 访问内存的次序安排及调整 264
10.4.1 访存次序和写缓冲器 265
10.4.2 实现 wbflush 266
10.5 用 C 语言开发 267
10.5.1 用 GNU C 编译器包裹汇编代码 267
10.5.2 存储器映射的 I/O 寄存器和“volatile” 268
10.5.3 用 C 语言开发 MIPS 应用时的其它杂七杂八的问题 270

第 11 章 MIPS 软件标准 (ABI) 272
11.1 数据表示和对齐 273
11.1.1 基本数据类型的宽度 273
11.1.2 “long”和指针类型的宽度 274
11.1.3 对齐要求 274
11.1.4 基本类型在内存中的布局和及其随尾端的变化 274
11.1.5 结构体和数组类型的内存布局和对齐 274
11.1.6 结构体中的位域 276
11.1.7 C 中非对齐的数据 278
11.2 MIPS ABI 的参数传递和堆栈约定 279
11.2.1 堆栈、子程序链接和参数传递 279
11.2.2 o32 中的堆栈参数结构 280
11.2.3 用寄存器传递参数 280
11.2.4 C 语言库函数中的例子 281
11.2.5 一个反常的例子：传递结构体 282
11.2.6 传递不定数量的参数 283
11.2.7 从函数返回一个值 284
11.2.8 寄存器用法标准的演化：SGI 的 n32 和 n64 285
11.2.9 栈的布局、栈帧以及对调试器的支持 287
11.2.10 个数不固定的参数和 stdargs 294

第 12 章 MIPS 调试——调试和剖析 296
12.1 “EJTAG”片上调试单元 298
12.1.1 EJTAG 的历史 299

12.1.2 怎样控制 CPU 300
12.1.3 通过 EJTAG 的调试通信 300
12.1.4 调试模式 300
12.1.5 单步运行 302
12.1.6 dmseg 存储器译码区域 302
12.1.7 EJTAG 的 CP0 寄存器，尤其是 Debug 304
12.1.8 DCR（调试控制）存储器映射寄存器 306
12.1.9 EJTAG 的断点硬件 307
12.1.10 理解断点条件 309
12.1.11 非精确的调试断点 310
12.1.12 EJTAG 的 PC 取样 310
12.1.13 无接探器使用 EJTAG 311
12.2 “EJTAG”之前的调试支持——断点指令和 CP0 观察点 312
12.3 PDtrace 313
12.4 性能计数器 314

第 13 章 天上掉下个林妹妹——GNU/Linux 316
13.1 基本概念 317
13.2 内核的分层结构 320
13.2.1 异常模式中的 MIPS CPU 320
13.2.2 关闭部分或全部中断的 MIPS CPU 321
13.2.3 中断环境 322
13.2.4 在线程环境中执行内核 322

第 14 章 软硬件怎样协同工作 323
14.1 中断处理的过程和时间 323
14.1.1 高性能中断处理和 Linux 325
14.2 线程、临界区和原子性 326
14.2.1 MIPS 体系结构和原子操作 327
14.2.2 Linux 回旋锁 328
14.3 系统调用时发生什么 329
14.4 MIPS/Linux 系统的地址转换 330
14.4.1 为什么要做存储器地址转换 332
14.4.2 基本的进程布局和保护 334
14.4.3 进程地址到真实存储器的映射 335
14.4.4 分页映射 336
14.4.5 我们真正想要什么 336
14.4.6 MIPS 设计的起源 338
14.4.7 跟踪被修改的页（模拟“Dirty”位） 341

14.4.8 内核对 TLB 重填异常的服务过程 342
14.4.9 TLB 的维护以及注意事项 345
14.4.10 存储器地址转换和 64 位指针 346

第 15 章 Linux 内核中关于 MIPS 的专门问题 347
15.1 直接管理高速缓存 347
15.1.1 DMA 设备存取 347
15.1.2 动态生成随后要执行的指令 349
15.1.3 高速缓存/存储器映射问题 349
15.1.4 高速缓存重影 350
15.2 CP0 流水线遇险 351
15.3 多处理器系统和高速缓存一致性 351
15.4 关键例程的手工调优 354

第 16 章 Linux 应用程序代码、PIC、库 356
16.1 链接单元怎样构成一个程序 358
16.2 全局偏移量表（GOT）的组织 359

附录 A MIPS 多线程 361
A.1 什么是多线程？ 361
A.1.1 同时运行两个线程需要哪些资源？ 362
A.2 为什么要用 MT？ 362
A.3 MIPS 怎样实现多线程？ 363
A.3.1 MT 新增的 CP0 寄存器 364
A.3.2 异常和中断 365
A.3.3 MIPS MT 和中断 365
A.3.4 线程优先级提示 366
A.3.5 用户权限的动态线程创建——fork 指令 366
A.4 MT 在实际中的应用 366
A.4.1 SMP Linux 367
A.4.2 用 MT 实现高速响应的程序设计 367

附录 B MIPS 指令集的其它可选扩展 369
B.1 MIPS16 和 MIPS16e ASE 369
B.1.1 MIPS16 ASE 中的特殊编码格式和指令 370
B.1.2 对 MIPS16 ASE 的评价 370
B.2 MIPS DSP ASE 371
B.3 MDMX ASE 372

MIPS 术语表 374

参考资料 **419**

译者补遗 **422**

序

MIPS 体系结构诞生于 1980 年代早期，最初来自于 John Hennessy 和他的学生在 Stanford 大学所做的工作。他们在探索 RISC (Reduced Intruction Set Computing) 体系结构的概念，该理论说相对简单的指令、结合优秀的编译器和使用流水线执行指令的硬件，可以在较小的面积上做出更快的处理器。这个概念是如此的成功，以至于在 1984 年专门成立了 MIPS Computer Systems 公司以实现 MIPS 体系结构的商业化。

在此后的 14 年间，MIPS 体系结构沿着几条不同道路发展演化，其实现非常成功地运用于工作站和服务器系统。在这期间，该体系结构及其实现经过增强后支持 64 位的寻址和运算，同时支持 Unix 等复杂的实现了内存保护的操作系统，以及高性能的浮点运算。也是在同一时期，MIPS Computer System 被 Silicon Graphics 收购，这样 MIPS 处理器成了 Silicon Graphics 计算机系统的标准。64 位处理器、高性能浮点运算、加上从 Silicon Graphics 继承的传统，使得 MIPS 处理器成了大批量销售的游戏机控制台所选择的解决方案。

1998 年，MIPS Technologies 从 Silicon Graphics 分离出来成为一家独立的公司，专注于针对嵌入式市场处理器的知识产权设计。结果是体系结构发展的步伐进一步加快，主要是针对于这个市场的特殊需求：高性能计算、代码压缩、几何图形处理、安全验证、信号处理和多线程。体系结构的每次发展都伴随有相应的实现该体系结构的处理器核，使得基于 MIPS 的处理器成为了高性能、低功耗应用的标准。

今天的嵌入式系统直接受益于 MIPS 在工作站、服务器等复杂系统中留下的遗产，因为嵌入式系统本身也变得越来越复杂了。一个典型的嵌入式系统往往由多个处理单元、一个高性能存储器以及一个或多个操作系统构成。与其它的嵌入式体系结构相比较的时候，MIPS 体系结构提供了一个已经实践证明可以在其上实现复杂系统的基础，而其它体系结构到现在才开始学习构建一个复杂系统所需要的东西。

在许多方面，《*See MIPS Run*》的第一版是关于 MIPS 体系结构及其实现的开创性的第一本书。其它的书也叙述类似的材料，但《*See MIPS Run*》专注于想要在 MIPS 上芯片有效的编程的程序员所要具备的体系结构和软件环境的知识。

嵌入式系统中不断增加的复杂度是通过对 MIPS 体系结构的增强来解决这类系统的需要的。对于任何一个当前在基于 MIPS 的嵌入式系统上工作的开发者，本书的第二版都是一本必读之作。书中增加了大量的新材料，包括 MIP32 和 MIPS64 对体系结构的标准化、多线程等全新的专用扩展，对于广为流行的 Linux 操作系统在 MIPS 体系结构上的实现也作了很好的处理。除了 MIPS 体系结构的规范之外，《*See MIPS Run*》第二版关于 MIPS 体系结构当前发展的资料是最新的；即使包括 MIPS 体系结构规范在内，该书仍然是所有资料中最具可读性的。

我希望你们能够和我一样觉得阅读本书既有价值、又有趣味。

Michael Uhler
Chief Technology Officer, MIPS Technologies, Inc
Mountain View, CA
May 2006

前言

这本书讲述的是 MIPS，在 1980 年代中期的那一批 RISC CPU 设计中曾经风靡一时。今天，MIPS 虽然并不是销量最高的 32 位体系结构，但却稳居第二。其成功之处，不用说，在于其应用的范围。围绕着 MIPS CPU 构建的设备可能只是价值 $35 美元的无线路由器，也可能是价值几十万美元的 SGI 超级计算机（当然随着 SGI 的破产，这些超级计算机的路已经走到了尽头）。介于在这两种极端之间，有 Sony 和 Nintendo 的游戏机、Cisco 的许多路由器、电视机顶盒、激光打印机等等。

这些年来，本书的第一版不仅卖出了近一万本英文本，而且还被翻译成了中文。这让我感到惊喜，我不知道原来外边还有这么多的 MIPS 程序员。

第二版的书名是《See MIPS Run ... Linux》。本书的第一版力图向读者展示 MIPS 体系结构的一些特性，因为如果不能看到这些特性怎样在操作系统内核内起作用，它们对读者就没有意义。但是现在许多读者已经对 Linux 怎样工作有些了解，我就可以直接引用其源代码。更重要的是，我引用源代码的时候，心里知道有些感兴趣的读者会阅读源代码，从中找出到底是怎么实现的。

所以虽然这是一本关于 MIPS 体系结构的书，但是最后的三章辗转于 Linux 内核和应用编程系统之间，其目的是阐明这些古怪的特性到底干什么用。我希望 Linux 专家能够原谅我对 Linux 细节的无知，但是难得有个机会讲述一下在一个真实的硬件体系结构上运行一个真实的操作系统，怎能容它错过。

MIPS 是一种 RISC：RISC 是个有用的缩写名词，适用于 1980 年代为实现高效的流水线而发明的一组体系结构的共同特性。CISC 这个词就有些含糊。我这里在狭义上使用这个词，用来指在 x86 和其它 1982 年之前的体系结构中发现的特性，设计时主要考虑的是用微代码实现。

有些读者可能要反对了：他在混淆体系结构和具体实现！没错。但是尽管计算机体系结构应当是和程序员之间有关什么程序能正确运行的协议，其自身也是一种工程设计。设计计算机体系结构是为了造出好的 CPU。随着芯片设计越来越复杂，这个界线也随着的不同的权衡在变。

本书是写给程序员的，这就是我们选材的标准——如果程序员可能会见到或者感兴趣，那些内容就会出现在本书中。这就意味着我们不会去讨论象曾经折磨了两代 MIPS 硬件设计工程师的奇怪的系统接口之类的问题。你的操作系

统可能掩盖了我们这里讲述的许多细节；有许多优秀的程序员认为 C 语言足够接近底层，还有可移植性的好处，关于体系结构的细节的知识无关紧要。但是有时候你的确需要下到硬件中去——而且人类生来就对于比特的世界的工作方式是充满好奇的。

这种定位取舍的结果就是，当描述一个软件工程师可能不熟悉的东西——特别是有关 CPU 内部的工作机制——的时候，我们倾向于采用一种非正式的方式。但是当碰到程序员接触过的材料，比如说寄存器、指令、以及数据在内存中的存储方式等问题时，我们则会采用简明扼要的专业术语。

我们假定读者熟悉和习惯了 C 语言。书中的许多参考资料用 C 程序片断来简化对操作的叙述，特别是在关于指令集和汇编语言细节的那几章。

本书的一部分针对接触过汇编语言的读者：MIPS 体系结构的简朴和特殊性从这一点上看来最为明显。但是如果汇编语言对你来说如同天书，那也不用担心天会塌下来。

本书的目标是告诉你对通用 MIPS CPU 上编程需要了解的一切知识。更确切的说，本书叙述体系结构时按照 MIPS Technologies 公司的 MIPS32 和 MIPS64 的定义——具体地说，是 2003 年发布的该规范的第二版。我们简称为“MIPS32/64”。但是本书并不仅仅是一本参考手册：要在头脑中对体系结构留下印象，就意味着必须全面理解。我也希望本书能引起正在面临选择专业的学生（在校的大学生或者刚刚入学的大学生）了解一种现代的 CPU 体系结构的兴趣。

如果你打算从头到尾按照章节次序阅读本书，期望从总体逐步过渡到细节，那么本书不会让你失望。但是你也会发现一些有关历史的发展；我们第一次讲到某个概念时，通常关注其第一个版本。Hennessy Peterson 称这为“从事物的发展历史中进行学习 (learning through evolution)，”当然对他们来说好的学习方法，对我来说也是。

第一章我们从一些历史背景讲起，把 MIPS 放到当时的时代背景下考虑，讨论其发明者心中最为关注的技术问题和思想。然后在第二章，我们沿着他们的思路方法讨论 MIPS 机器语言的特性。

为了帮助读者建立一个总体概念，我们在第三章之前都略去了处理器控制的细节。第三章介绍了难看但却极为实用的处理器控制系统，MIPS CPU 要通过它来处理高速缓存、异常及启动、以及内存管理。上面的三个专题，分别构成了第 4 到 6 章的内容。

MIPS 体系结构非常小心的分离出了处理浮点数的指令集部分。这种分离允许制造的 MIPS CPU 带有各种不同程度的浮点支持硬件，从根本没有到部分实现一直到最先进的。我们也把浮点功能单独分开，留到了后面的第 7 章。

到这点为止，每一章都遵照一个合理的次序逐步介绍 MIPS。后面的几章改变了方向，更像一个参考手册，或者基于例子的学习教程。

第 8 章，我们全面考查整个机器指令集；试图在准确的同时又要比标准

MIPS 参考手册简要得多——我们只用了 10 页来介绍而在别处花了一百多页的内容。[1] 第 9 章简要的介绍了如何阅读和书写汇编语言，但远远够不上一个汇编语言编程手册。

第 10 章是为那些不得不在别的 CPU 和 MIPS CPU 之间移植软件的人提供的有关移植问题的一览表，同时附带了一些有益的建议。最长的一节讲述 CPU、软件和系统中的尾端导致的各种问题及解决方法。

第 11 章对为了生成可以在不同工具包互操作的软件而必须的软件约定（寄存器用法、参数传递等）作了一个简要的总结。第 12 章介绍了在 MIPS CPU 上标准化的调试和剖析特性。

接下来我们上路去看看 MIPS 怎样运行 GNU/Linux。我们在第 13 章先讲解 Linux 内核和计算机体系结构之间的关系；然后第 14 和 15 章深入某些细节具体看看 MIPS 体系结构是怎样满足 Linux 内核的需要的。第 16 章快速介绍一下使得 GNU/Linux 应用程序工作起来的动态链接机制。

附录 A 讲述了 MIPS MT(multithreading 多线程) 扩展，这可能是许多年来对体系结构最重要的扩展。附录 B 描述了重要的附加部分：MIPS16、新的 MIPS DSP 扩展以及 MDMX。

在本书的结尾，你还可以看到一个词汇表——是个查找不熟悉的或者专业的用词和缩写的好地方——还有一个参考书、论文以及供进一步阅读的网上参考资料的列表。

风格和局限

每本书都反映了作者的思想，所以我们最好谈一下本书的思路。

因为你们当中有些是学生，我曾经想要不要区分 MIPS 的用法和一般的用法。我最后决定不这样做。我希望能够有针对性，除非不用代价就能推广到一般情形。我也试图尽量具体而不要抽象。我并不太关心象“TLB”这样的术语在更大的范围内是什么意思，但我会解释在 MIPS 环境中代表什么。人类是很善于总结推广的，这样做不会有损于你们的学习。

自从 1986 年秋季我开始从事 MIPS CPU 方面的工作以来已经有 20 年了。本书中的有些材料要追溯到 1988 年，当时我刚开始进行 MIPS 体系结构的培训课程。1993 年，我把这些材料收集整理起来，制作成一本针对 IDT 的 R3051 系列 CPU 的软件手册。此外还补充了相当多的额外材料来才构成了本书的第一版，于 1999 年出版。

自从 1999 年以来发生了许多事件。现在 MIPS 在 SGI 的服务器中的生命已经走到了头，但在嵌入式系统中杀出一条血路，并且抢占了重要的一席之

[1] 在生成这些表的时候，我花费了相当大的精力仔细检查过，表中绝大多数内容是对的。但是如果你的系统要用到它，一定要交叉检验这些信息。一个相当可靠的很好的信息来源可以从 GNU 工具包的行为和源代码中找到——但是我也参考了这里，所以不是完全独立的。

地。Linux 成为了嵌入式 MIPS 用得最多的操作系统，但是在嵌入式市场仍然呈现多样化的局面。MIPS 的规范也围绕着 MIPS32 和 MIPS64（本版以它作为基准）进行了重新组织。本书的第二版已经写了三年了。

MIPS 的故事还在继续；要不然，本书只能写给历史学家，Morgan Kaufmann 也不会出版本书有太大的兴趣了。截至 2005 年底还未发布的 MIPS 的进展，对于本书这一版来说因为实在太晚了而未能涉及。

排版约定

本书使用的字体约定如下：

- 这种字体表示正文。
- 这种字体表示边栏正文。
- **这种字体表示汇编代码和 MIPS 寄存器的名字。**
- 这种字体表示 C 语言代码和十六进制数。
- 这种字体表示硬件信号名字。

致谢

本书的主题一直跟随着我的计算生涯。Mike Cole 让我对计算产生了兴趣，我一直在模仿他的技巧挑选出好的思想。在 WhiteChapel Workstations 那段短暂而又充满激情的的日子里，许多同事教给我计算机体系结构和硬件设计的知识——教我最多的要属 Bob Newman 和 Rick Filipkiewicz 了。我也要感谢 WhiteChapel 的销售员 Dave Gravell, 是他让我最初接触了 MIPS。对我在 Algorithmics Ltd 的同事（Chris Dearman、Rick Filipkiewicz、Gerald Onion、Nigel Stephens、Chris Shaw）不得不致以双倍的感谢，一是为我在和他们的无数次的探讨、争论、设计中所学到的一切东西，二是为他们对写作本书占去我本该和他们在一起的时间所给与的理解。

太多的感谢要给长期以来阅读本书各章节的审稿人：Integrated Device Technology 的 Phil Bourekas、LSI Logic Corporation 的 Thomas Daniel; Silicon Graphics 的 Mike Murphy、Carnegie Mellon University 的 David Nagle。

关于第二版要的一些说明：因为都在 MIPS 公司周围工作，我认识 Paul Cobb 很久了。Paul 贡献了许多材料更新了 MIPS CPU 的历史综述，还有编程的一章，并且整理了参考文献引用。在所有情况下，最后都是由我进行编辑的——所以任何错误都应该属于我。

在准备这一版的过程中，我就职于 MIPS 公司。我知道只挑出一些同事表示感谢而没有提及其他同事是件危险的事情，但是不管怎样我还是决定要这样做。

Ralf Baechle 在运作 `www.linux-mips.org` 站点，协调 MIPS 和 Linux 内核上的工作。他对于打消我对 Linux 的一些错误认识帮助很大：我一开始认为 Linux 和别的操作系统差不多...（Robert Love 的《Linux Kernel Development》一书也很有帮助；我向任何想更多了解内核的人热情推荐本书）。感谢 MIPS 公司和我的各级经理和领导，他们允许我灵活的安排工作时间，还要感谢很多在 MIPS 公司的同事（太多了而无法一一列举姓名）阅读和评价了本书的草稿。

Todd Bezenek 是我永远的同事，也是这一版的审稿人。在 MIPS 公司之外还有一些审稿人出于对这个领域的爱好和兴趣审阅了本书：主要的贡献者有 Steven Hill (Reality Diluted, Inc)、Jun Sun(DoCoMo USA Labs)、Eric DeVolder 和 Spphie Wilson。

Denise Penrose 是当之无愧的最佳编辑。在 Finsbury Park（我家所在的北伦敦）能够说他们正要飞往 San Francisco 和出版商共进早餐的人不多。

最后还要说的重要的一点就是，感谢 Carol O'Brien，在我重写本书的中间不顾一切地嫁给了我。

第 1 章　MIPS 和 RISC 体系结构

MIPS 是实际使用的 RISC 体系结构中最精巧的一种；即使连竞争对手也这样认为，这可以从 MIPS 对于后来的体系结构比如 DEC 的 Alpha 和惠普的 Precision 产生的强烈影响看出来。在一个充满竞争的市场上，仅仅靠精巧的设计本身并不能保证长盛不衰，但是 MIPS 微处理器却常常能通过保持设计的最简化在每一代处理器技术发展中都跻身性能最佳的行列。

相对简单的设计对于 MIPS 计算机系统公司来说来说出于一种商业需要，该公司是 1985 年从一个学术研究项目中分离出来专门从事制造和销售芯片的。结果该体系结构在当时（现在可能依然）得到了工业界大范围内的有影响力的制造商们的广泛支持。从生产专用集成电路核心(ASIC Cores)的厂家(MIPS Technologies，Philips)，到生产低成本 CPU 的厂家(IDT、AMD/Alchemy)，到唯一得到广泛使用的 64 位嵌入式处理器的生产厂家(PMC-Sierra、Toshiba、Broadcom)都有。

在低端，片上系统 SOC 的 MIPS CPU 小到人的肉眼几乎难以看清，而在高端，Instrinsity 的性能出众的处理器可以运行于 2 GHz 主频——这个速度，在除了不计较功耗/发热的当代 PC 处理器之外的领域中，可以说是无与伦比。

ARM 要比 MIPS 更出风头，但是 MIPS 的销售量也够好的：2004 年有一亿片的 MIPS CPU 投放到了市场用于嵌入式领域。

MIPS CPU 是 RISC 结构 CPU 的一种, 它诞生于一个学术研究发展硕果累累的时期。RISC(Reduced Instruction Set Computer ——精简指令集计算机)是一个富有吸引力的缩写，与很多这样的缩写一样，字面的缩写可能更多地掩盖了而不是揭示了其内涵。但是 RISC 作为一个简洁通用的名称，在当时为泛指那些在 1986 到 1989 年之间投放市场的新型 CPU 体系结构确实起到了一定作用。这些体系机构的优异性能都源于数年前的几个具有开创性的研究项目所提出的思想。有人曾说："RISC 就是任意定义于 1984 年以后的计算机体系结构"；这句话本意是在嘲讽工业界对这个概念的滥用，但是从技术角度来看，它也确实反映了这样一个现实——没有任何一款 1984 年以后定义的计算机能够忽视 RISC 开创者们的工作。

这些具有开创性的项目的其中一个就是斯坦福大学的 MIPS 项目。该项目命名为 MIPS (Microcomputer without Interlocked Pipeline Stage ——无互

锁流水阶段微型计算机——的缩写)，同时也和“million instructions per second——每秒百万条指令数”的缩写一语双关。斯坦福研究小组的工作表明虽然流水线已经是一种众所周知的技术，但是以前的体系结构对它的利用远远不够，其实流水线技术本来完全可以利用得更充分，尤其是和 1980 年的半导体设计技术相结合时。

1.1 流水线

从前在英格兰北部的一个小镇里，有一家名叫 Evie 的人开的卖炸鱼和炸薯片的店。在店里面，每位顾客买想要的食物（通常是炸鳕鱼，炸薯片，豌豆粥，[1]和一杯茶）需要先排队，然后等着盘子装满后坐下来进餐。

Evie 店里的炸薯片是小镇中最好的，在每个集市日，排队买午餐的队伍都会一直排到店外。所以当隔壁的木屐店关门的时候，Evie 就把它租了下来并加了一倍的桌椅。但是仍然不能容纳下所有的顾客。外面排着的队伍还是和从前那么长，忙碌的小镇居民都没有时间坐等茶凉了。

他们没地方再加服务柜台了；Evie 的炸鳕鱼和伯特的炸薯片占满了空间。但是后来他们想出了一个聪明的办法。他们把柜台加长，Evie、Bert、Dionysuss 和 Mary 站成一排。顾客进来的时候，Evie 先递上一个盘子并盛上炸鳕鱼，然后 Bert 给加上炸薯片，Dionysus 再给盛上一勺豌豆粥，最后 Mary 倒茶并收钱。顾客们排队向前移动；当一个顾客盛到豌豆粥的同时，后面的一个已经盛到了炸薯片，再后面的一个刚刚盛到鱼。少数穷苦的居民不吃豌豆粥——那也没有什么，他们不用从 Dionysus 那里盛东西，只从他那里得到一个浅浅的微笑。

这样一来队伍变短了。不久以后，他们又买下了另一边隔壁的店铺，增加了更多的餐位。

这就是流水线。将那些重复性的工作按顺序分成几个步骤，合理安排使得工作依次通过每个工人，每个工人只需要依次完成自己的那部分工作就可以了。虽然单个顾客等待服务的总时间增加了，但是却有四个顾客能同时得到服务，这样在集市日的午餐时段里能够接待的顾客总数增加了三倍。图 1.1 说明了 Evie 的方法，是她的儿子 Einstein 在少有的一次过问非虚拟现实的时候画的[2]。

如果将程序看成是内存中存储的一堆指令的话，准备运行的程序看起来和排队等待服务的顾客没什么相似之处。但是如果从 CPU 的角度来看，情况就不一样了。CPU 从内存中取出一条指令，进行译码，找到需要的操作数，执行相应操作，最后保存产生的结果——然后再次重复同样的工作。等待执行的程序就是一个指令队列，它们一次一个的流过 CPU。

由于每条指令要做不同的工作，因而在 CPU 内部已经配有各种不同的专用的逻辑模块，所以构造一个流水线并没有给 CPU 额外增加多少复杂度；只是让

[1]非英语读者不必深究其中的微妙含义。[译者注]豌豆粥通常要和前两者一起吃。
[2]听起来给我感觉她的儿子好象是在看计算机科学的书。

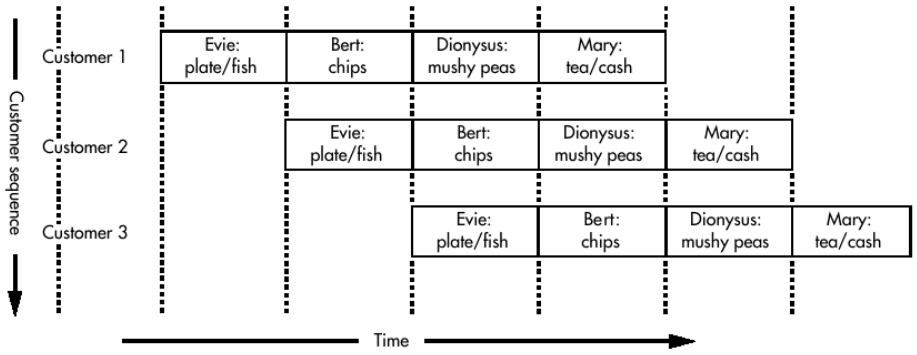

图 1.1: Evie 店里的流水线

CPU 更接近全速运行而已。

使用流水线技术不是 RISC 微处理器的创新。关键的地方在于一切从头开始重新设计——从指令集开始——以使提高流水线的效率[3]。那么，怎样才能使流水线效率提高呢？实际上是这种问法本身可能就错了。正确的问法应该是，什么使得流水线效率低下了？

1.1.1 什么使得流水线效率低下了？

在流水线中一个阶段比其它阶段占用时间过长并不是什么好事。Evie 的小店的组织依赖于 Mary 一手倒茶、一手给人找钱的能力——如果 Mary 比别人慢，整个队列就得慢下来等她。

在流水线中，要尽量确保每个阶段花费大致相同的时间。电路设计经常会碰到要在逻辑的复杂性和速度之间进行权衡的机会；设计者可以把任务分配给不同的阶段；仔细设计各个阶段使流水线保持平衡。

问题困难的地方不在于动作有多么复杂，而在于有些难缠的顾客。回到上面店中的例子，由于 Cyril 手头经常缺钱，所以 Evie 要等到 Mary 点完他的钱，才能给他盛炸鳕鱼。当 Cyril 排到 Evie 那里时，他就要停下来等待，一直等到 Mary 服务完前面三位顾客然后把他的一堆旧得有点变形的硬币点清楚。这样 Cyril 就出问题了，因为他要用到前面的顾客正在占用的资源（需要 Mary 点钱），就是说发生了资源冲突。

Daphne 和 Lola 总是一起来并且分享他们的食物。如果 Daphne 不买茶的话，Lola 就不会买炸薯片（太咸没有喝的吃不下）。Lola 将会在排到 Bert 那里时驻足不前，一直等到 Daphne 排到 Mary 那里，这样流水线出现了一个空隙。这

[3]这个意义上的第一个 RISC 可能是 Seymour Cray 于 1970 年代设计的 CDC6600，但当时这种思想还不流行。这已经有点离题跑到计算机体系结构的历史上去了，对此感兴趣的读者可以参阅 Hennessy & Patterson 1996 。

就是依赖(相应的空隙称为流水线空泡——*pipeline bubble*)。

并非所有的依赖都有问题。Frank 总是和 Fred 买完全一样的食物，但他只要跟在 Fred 后面就行——如果 Fred 买了炸薯片，那 Frank 也要炸薯片。

要是能够排除麻烦的顾客，就可以得到更高效的流水线。这对 Evie 来说不是一个可行的做法，因为她需要在这个由形形色色的人们所组成的小镇上谋生。Intel 公司面临同样的问题，它的处理器的吸引力依赖于用户能够继续运行所有那些老的软件。但是如果是新的可以从头开始定义指令集的 CPU，你就可以排除许多这样麻烦的事情。在 1.2 节，我们会看到 MIPS 是怎样做的。现在我们先回到一般的计算机硬件上来讨论一下高速缓存。

1.1.2 流水线和高速缓存

我们前面提到，高效的流水线操作要求每个阶段占用相同的时间。但是一个 2006 年的 CPU 计算两个 64 位数加法的速度比它从内存读取数据的速度要快 50 到 100 倍。

所以有效的流水线还依赖于另一种能够将大多数内存访问的速度提高 50 倍的技术——使用高速缓存（常简称缓存）。高速缓存是一种可以高速存取的小容量的局部存储器，其中保存着内存数据的拷贝。每一数据单元都保存了其在内存中的地址（即高速缓存标签）。当 CPU 需要数据的时候首先查找高速缓存，如果找到可用的数据就可以快速送回。因为无法猜测 CPU 下一步需要什么样的数据，高速缓存只是保存 CPU 最近用到过的数据；当有更多的数据从内存到达时，缓存中的一些数据需要丢弃以便腾出空间。

即使一个简单的高速缓存也能在 90% 以上的时间提供 CPU 所要用需的数据，所以流水线设计仅仅需要留出足够从缓存中存取数据的时间就行了；缓存未命中发生的机会相对较少，确实发生时，我们可以让 CPU 停下等待一会儿（聪明的 CPU 能够利用这段时间见缝插针地找些有用的工作来做）。

MIPS 体系结构设计了分离的指令和数据高速缓存，所以它可以在取指令的同时读写数据。

CISC 体系结构也有高速缓存，但是高速缓存常常是作为存储器系统的一个特性后来加上去的。RISC 体系结构更适合于把高速缓存看作为 CPU 的一部分，并和流水线紧密相关。

1.2 MIPS 的五级流水线

MIPS 体系结构是为流水线而设计的，图 1.2 接近于最早期的 MIPS CPU 和许多典型的 MIPS 流水线。只要 CPU 从高速缓存中运行，每条 MIPS 指令的执行过程就分为五级，每一级称为一个流水线阶段，每个阶段占用固定的时间。这个固定的时间通常就是一个处理器时钟周期（但有些操作只用半个时钟周期，所以 MIPS 的五级流水线只用了四个时钟周期）。

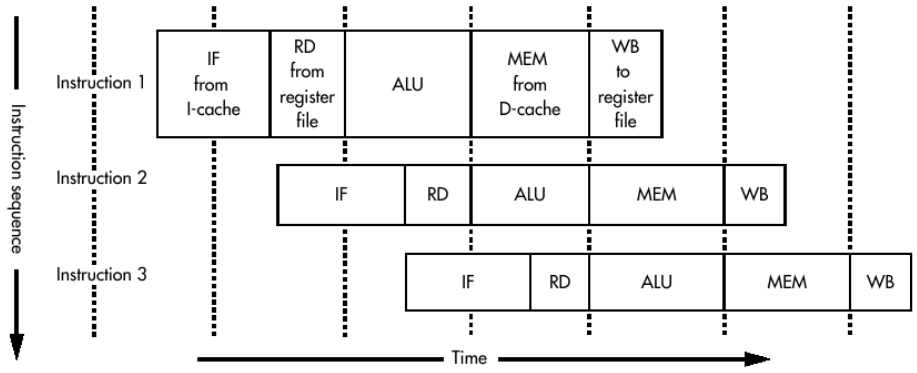

图 1.2: MIPS的五级流水线

所有的指令都经过严格定义，即使在流水线的某一级没有任何操作也要遵循同样的流水线序列。最终的结果就是，只要保持高速缓存命中，CPU 就能在每个时钟周期完成一条指令。

我们对照图 1.2，讨论一下每个流水阶段发生的事情。

- IF 取指(insturction fetch)，从指令高速缓存（*I-cache*）获取下一条指令
- RD 读取寄存器(read register)，读取该指令的源寄存器域指定的 CPU 寄存器的内容。
- ALU 算术逻辑单元(arithmetic/logic unit) 在一个时钟周期内完成算术或者逻辑操作（浮点运算和整数乘除无法在一个时钟周期内完成，对其处理有所不同，后面会讲到这一点）。
- MEM 访问内存（memory）该阶段指令可以读写数据高速缓存（D-cache）中的内存变量。平均而言，每四条指令中就有三条指令在该阶段没有任何操作，但是为每条指令都分配了这一阶段以确保不会出现两条指令同时需要访问数据高速缓存的情形（这好比上面例子中 Dionysus 打的豌豆粥）。
- WB 写回寄存器（write back），将操作结果值写到寄存器堆中。

你也许见过别的看上去略有不同的 MIPS 流水线图。习惯上画的时候为了方便，把每个流水线阶段画的看上去好像都恰好用了一个时钟周期。后来的一些 MIPS CPU 的流水线更长或者有轻微的改动，但是这种四个时钟周期的五级流水线是 MIPS 体系结构最初所用的，而且在一些简单的 MIPS CPU 中现在依然用的是与此非常相似的流水线。

严格的流水线要求限制了指令的某些操作。首先，它要求所有的指令要一样长（正好一个机器字长 32 位），这样取指时间就可以是常数。这本身就不支持

复杂的指令；例如指令中没有足够的位用来编码太过复杂的寻址模式。而且固定长度的指令直接导致一个问题；在象 x86 这样的体系结构上的典型程序中，平均的指令长度为三个字节多一点。MIPS 的代码要占用更多的存储空间。

其次，流水线设计排除了实现操作内存变量的指令。直到流水线的第四个阶段才从内存或高速缓存读写数据，这对 ALU 来说实在太晚了。内存访问仅仅用一个简单的 load 和 store 的指令在内存和寄存器间传送数据（后面将看到这称之为 *load/store* 体系结构）。

1987 年投放市场的 RISC CPU 工作得很好，是因为在这些限制下设计的指令集被证明一点都不比那些复杂的指令集差（特别是对于编译出的代码），而后者给硬件带来太多的痛苦。1987 年及其后的 RISC 有些共同的特征，就是围绕高效的流水线而设计的指令集和高速缓存的使用。

MIPS 项目组的体系结构专家也参与了当时关于具备什么特征的 CPU 适合高效优化的编译器目标的思考。其中很多需求和流水线的需求是兼容的，所以 MIPS CPU 设计有 32 个通用寄存器和三操作数的算术/逻辑指令。幸运的是，那些打乱流水线的复杂的专用指令也常常是编译器不愿产生的。

RISC 先驱者们的判断经受住了时间的考验。最新的指令集已经将硬件/软件的分界线向后更推进了一步；称为超长指令字 VLIW(very long instruction word) 或者显式并行指令计算 EPIC(explicitly parallel instruction computing)。其中最显著的就是 Intel 的 IA64 体系结构，尽管投资巨大却并不成功；看来他们是弄错了硬件/软件的边界。

1.3 RISC 和 CISC

我们现在可以尝试一下定义这两个已经多次用到的术语了。对我来说，RISC 是一个用于形容机器的体系结构/指令集的形容词。在 1980 年代中期，它用于指一组相对较新的、其指令集做了精巧和有效的规定以使得流水线能够成功和高效实现的体系结构。由于 SPARC、MIPS、PowerPC、HP Precision、DEC Alpha, ARM(在相对较小的程度上) 明显存在的高度相似之处，使得 RISC 成为一个很有用的概念。

与这个相当有明确针对性的褒义的描述形成对照的是，CSIC（Complex Instruction Set Computing）用来带有贬义的描述那些没有按照关于流水线实现的思想设计的体系结构。RISC 体系结构的革命是如此的成功，以至于 1985 年后的体系结构没有哪一个背离了基本的 RISC 原则[4]；因而，CISC 体系结构必然指的是那些诞生于 1985 年之前的体系结构。在本书中，你可以合理的认为关于 CISC 的东西既适用于 Intel x86 家族处理器，也适用于 Motorola 的 680x0 系列。

[4] 即使是 Intel 的充满各种复杂创新设计的 IA64 也具有 RISC 的流水线友好的特征。但是 Intel 用的形容词 EPIC 更能反映他们的宏伟目标以及重大失败的可能性。

当不是用来指指令集而是用于指具体实现的时候，对这两个术语都是一种讹用。可以肯定的是 Intel 通过应用 RISC 先驱们的实现技巧来提高了其和 RISC 结构相差很远的 x86 家族处理器的性能。但是要把这些实现说成是拥有 RISC 核则是一种误解。

1.4 过去和现在的一些重要的 MIPS 芯片

现在让我们花点时间游览一下在过去 20 多年来 MIPS 处理器以及使用它的系统的发展历程/演化历史。我们大致按照事件发生的时间顺序游览，沿途偶尔绕道造访几处名胜景点。一路上我们将看到尽管 MIPS 体系结构一开始设计是冲着 Unix 工作站方向前进的，但从那以后找到了通向各种各样应用的道路——其中许多在最初几年是根本不可想象的。在接下来的几章里，你将会对下面提到的这些名字有更多的了解。

尽管指令集和硅半导体技术已经发生很多变化，但是 1985 年的 R2000 上的用户级软件在任何现代的 MIPS CPU 上都能十分高效的运行而没有一点问题。这可能是在所有主流的体系结构中向后兼容性做得最好的了。

1.4.1 R2000 到 R3000 处理器

MIPS 成立为一家公司

MIPS 计算机系统有限公司成立于 1984 年，以便将斯坦福的 MIPS CPU 实现商业化；我们以后就简称 MIPS 公司。斯坦福大学的 MIPS 只是几个以全新的方式集成了芯片设计、编译优化和计算机体系结构为一体的并取得成功的美国学院项目之一。商业化的 MIPS CPU 加上了内存管理单元，于 1985 年后半年以 R2000 首次面世。

即使在 1980 年代中期，建立芯片制造工厂也十分昂贵；这显然超出了一个新建的小公司的财力。MIPS 公司通过向现有的已经投入了巨资的半导体厂商授权制造的方式让其设计变成产品。早期的授权生产商包括有 IDT、LSI Logic、Performance Semiconductor 和 NEC。

1987 年中期首次加入了功能强大的片外数学协处理器（R2010 浮点加速器）。因为 MIPS 公司想要进入工程工作站领域这个潜力巨大的市场，良好的浮点性能必不可少，R2010 及时提供了这一点。

MIPS 自己买了一些这些厂商制造的芯片，把它们集成到自己的小型服务器和工作站中去。这些厂商也可以根据授权协议向别的客户供应芯片。

R3000 处理器

首次于 1988–1989 年面世，其采用了更为先进的制造工艺和一些经过仔细筛选的硬件增强措施，二者的结合造就了性能的大幅提升。从编程的角度来看，

几乎和 R2000 没有差别，这就意味着新设计的速度提升可以立刻从快速增长的软件上体现出来。不久以后又加入了 R3010 FPU ——与其上一个版本做了类似改进。

到 1990 年代中期，一些先行者开始在嵌入式应用中使用 R3000, 一开始时在高性能激光打印机和排版设备中使用。

R2000/R3000 芯片内有一个高速缓存控制器——要用高速缓存，只要加上 SDRAM 就行了。FPU 在（与整数 CPU 并行）读取指令、传送操作数和运算结果时共享高速缓存总线。对于 1986 年的 CPU 速度，这样的分配功能是新颖、实用和可行的；更重要的是，这种做法让每个芯片上的信号连接数保持在当时通用的封装技术所能达到的引脚数的限制之内。这使得可以在合理的成本上生产芯片，并能采用现有的生产设备组装成系统。

提高时钟频率的挑战

尽管这种设计在当时是合理的，但是最终问题还是出在 R3000、 R3010 FPU 和外部高速缓存的的功能分配上。

首先外部高速缓存的实现间接导致了系统设计的一些难题。为了从外部缓存的 RAM 中挤出尽可能高的性能，其控制信号就必须在极端精确的非常短的时间隔内实现切换。实现精确延时的责任被传递到系统设计师身上：R3000 需要四个由外部产生的输入时钟信号的不同相移的拷贝，这个相移定义的时间间隔对于缓存控制信号的正确管理是至关重要的。在 20 兆赫的时候这还可以对付，但是当时钟频率体高到 30 兆赫以上时，对精度的要求就已经极为苛刻了，使得时钟管理的任务极为困难。

其次，提高系统时钟的压力也给 RAM 厂商带来了问题：为了与处理器流水线中不断缩短的时钟周期时间跟上步伐，他们不得不想方设法在内存芯片存取时间方面的做出相应的改进。

所有这些问题在 1980 年代末期的时候越来越明显，其限制使得这一代的设计只能做有限幅度的改进。从 1988 年的 25 兆赫开始，R3000 系统好不容易在 1991 年才达到了 40 兆赫——就再也没法提高了。

1.4.2 R6000 处理器：一度转向

1980 年代后期出现了处理器设计者之间的关于提高微处理器时钟频率的争论。特别是有两个问题浮出了水面。第一个：未来的处理器设计将高速缓存放在外部实现好呢，还是把高速缓存做到芯片上好？第二：未来的处理设计选择哪种逻辑技术最有利？

第一代 RISC CPU 采用的是 CMOS 芯片。它们运行良好，把许多逻辑封装到很小的空间，而且所有的低成本（pre-RISC）微处理器都采用了 CMOS。CMOS 的支持者认为在未来的许多年内依然拥有这种优势。不

错，CMOS 逻辑不是最快的，但是这一点可以克服——象 Intel 这样的公司肯定很快就会进行相应的必要的投资。而且他们认为 CMOS 在已经做得很好的方面还有提高的空间——在给定的硅片面积上集成更多的逻辑门、在给定功率下实现更高的开关频率。

另一些设计者觉得对 CPU 来说提高速度是多么的迫切，他们得出结论说高端的处理器最好采用 ECL 芯片，就象那些已经在大型机和超级计算机 CPU 采用的那样。简单的 ECL 逻辑门速度更快，在芯片之间发送信号的速度更是要快得多。但是，单个芯片上集成的逻辑门数量少，而且运行时发热量大。

鉴于这两种技术各自面临不同的问题与挑战，很难预测哪一个最后将胜出。Bipolar Integrated Techonlogy(BIT)公司是 ECL 的支持者。1988年它开始了称为 R6000 的 MIPS CPU 的工作。该项目雄心勃勃，BIT 希望能象 CMOS RISC 微处理器曾经改写工作站的性能那样，由自己改写“超级小型机”的性能。

但是他们碰到了问题。因为 ECL 的密度限制，处理器设计不得不分成多个芯片。客户对于完全转向 ECL 的芯片到芯片信号标准心存疑虑。BIT 构造了 BiCMOS 混合体以寻求结合这二种技术的最好的优点。

此后，越来越多的问题拖垮了该项目。R6000 项目被一个又一个的问题拖延，最终拖到了 R4000 芯片之后：R4000 是第一个利用高集成度把高速缓存移到片上的新一代 CMOS 芯片，以另一种途径提高了时钟速度。

BiCMOS CPU 并没有随着 BIT 的倒闭而死亡：几年之后，一个名为 Exponential Technology 的公司做了大胆的尝试，在 1996 年前后做出一个 PowerPC 的 BiCMOS 实现，其时钟频率超过了 500 兆赫，这在当时令人刮目相看。然而和 BIT 一样，该公司因为技术和合同方面的双重挫折最终走向倒闭。

在大的方面，双方都犯有错误。最后，过去了几年的时间，片上高速缓存技术才成为达到最高时钟速度的必不可少的条件。惠普公司坚持 CMOS 技术和（大的）外部高速缓存为它自己的很象 MIPS 体系结构的 Precision 做开发。惠普最终将其时钟频率提高到了 120 兆赫左右——是最快的 R3000 的三倍——并没有采用 ECL 或者 BiCMOS 。惠普自己做自己的客户，把这批处理器用到自己的工作站上。该公司认为这个市场最好由一种突破性的方法，加上能够承受高引脚数封装技术以及细致高速的系统级设计成本的技术来推动。这种策略在很长一段时间里把惠普推上了处理器性能的顶峰；赢家不总是属于最激进的变革。

1.4.3 第一个 CPU 核

在 1980 年代早期，LSI Logic 最早提出了采用批量生产的芯片设计和加工技术让系统公司能够为其自己产品的需要量身定做芯片的思想。这些芯片称为专用集成电路——Application Specific Integrated Circuits（ASIC）；到 1990 年前后，ASIC 可以包含多达几千个逻辑门，相当于一个大的电路板上布满了 1970 年代的逻辑设备。单片的成本很低，开发的成本也可以控制。

我们已经说过了，LSI 公司很早就对 MIPS 感兴趣，而且制造了一些

R2000/R3000 的芯片。两年以后，该公司很自然的转向了用他们自己的 ASIC 技术来实现 MIPS 体系结构；这一转向打开了通向希望在一个芯片中同时包括一个 MIPS 处理器和其它的逻辑模块的客户的大门。别的 MIPS 的授权生产商，比如 IDT，也开始提供同时内置一个 MIPS 处理器和集成简单外围功能的产品。

即便在 1990 年代初期，你也能很容易的把一个 R3000 级别的 CPU 的基本逻辑放到一个 ASIC 上；但 ASIC 没有很高效的 RAM 模块，所以集成高速缓存是个问题。但是 ASIC 发展很快，到 1993 年在一个芯片上实现完整的微处理器系统——不光是 CPU 和高速缓存，还有内存控制器、接口控制器以及其它各种小的支持逻辑模块——的想法已经可以实现。

ASIC 的业务依赖于客户能够在相对较短的时间——远远短于用“定制”所需要的时间——内将设计投入生产。尽管向客户提供将整个系统集成到单个芯片上的想法非常有吸引力，ASIC 厂商还得要处理好一个平衡：怎样既能在保持设计周期在客户以前期望的范围内的同时，又要能适应无法回避的设计复杂度增加？

ASIC 行业的答案是以核的形式提供有用的功能部件——比如一个完整的 MIPS 处理器。核就是现成的封装了所有必要的内部设计工作和验证逻辑的的模块，通常以一组 ASIC 设计软件能接受的格式的二进制文件的方式呈现。未来的系统设计将通过把几个 ASIC 核连到一个芯片上；与现有系统——连接电路板上的元件——相比，实现为基于核的 ASIC 将会更小、更快、更便宜。

此前，ASIC 设计者思考时很自然的想到组合小的逻辑模块——状态机、计数器、解码器等等。随着ASIC 核的出现，设计者更愿意在宽的画布上用更粗的画笔来描绘，把处理器、内存、内存控制器以及片内总线集成到一起。

如果你怀疑事情并不那么简单，那说明你的直觉是对的。这东西听起来很吸引人，但实际做起来，创建核以及把核连接起来都是非常难办的事情。不管怎样，这些早期的 ASIC 核仍然具有重要的历史意义；他们直接衍生了后来在 2000 年代初期普及的片上系统 SoC 设计。我们介绍完整个 1990 年代的 MIPS 发展的几条线索之后，稍后将回头再讲 SoC 的故事。

R4000 处理器：一次革命

1991 年面世的 R4000 处理器，是一个大胆和突破性的发展。其领先的，开创性的特性包括一个完整的 64 位指令集、当时最大的片上高速缓存（两个 8 K），在当时高得就象是在科幻小说中的才有的时钟速度（发布时为 100 兆赫），一个片内二级缓存控制器，一个以系统内部时钟分频运行的系统接口，片内自带的支持共享存储多处理器系统的逻辑。R4000 是首批采用了诸多到 1995 年才广泛使用的许多技术的芯片，也要注意很重要的一点是它并未采用复杂的超标量执行。

R4000 并非完美，它是一个功能强大的芯片，其设计很难测试，特别是其采用的支持多处理器的复杂的技巧。与 R3000 相比，它需要更多的时钟周期来执行给定的指令序列——这些时钟周期短到以致过早结束，但是你并不愿意损失

性能。为了赢得时钟速度，一级缓存做到了片上：为了使得每个芯片的成本保持合理，一级缓存的容量不得不保持相对较小。R4000 的流水线更长，主要是想把缓存访问分解到多个时钟周期中。更长的流水线导致效率降低，因为当流水线被分支指令打乱时损失了更多时间。

ACE 联盟的起落

在 R4000 面世前后，MIPS 公司曾高度期望新的设计能够帮助它在工作站、桌面系统和服务器市场赢得一席之地。

这可不仅仅是 MIPS 公司的一厢情愿的想法。在 1990 年代初期，许多观察家们预言 RISC 处理器将会从其 CISC 的竞争对手那里不断占领更多的市场；甚至有更大胆的预言家表示 CISC 家族的处理器将在几年之内彻底消失。

1991 年，大约 20 个公司成立的作到一起成立了一个名为高级计算环境 ACE（Application Computing Environment）的联盟。该组织成员包括 DEC（小型机）、Compaq（PC）、微软和 SCO（当时负责 Unix System V）。ACE 的目标是定义规范和标准以便让未来的 Unix 或 Windows 软件直接运行在任意一款采用 Intel x86 或 MIPS CPU的机型上。即使在 1991 年，PC 市场业务的一小部分对 MIPS CPU 和 MIPS 系统厂商也意味着很可观的销售额。

如果炒作能够创造成功的话，那么 ACE 定当是其中的佼佼者。但是现在回头来看，微软当时更感兴趣的是只是证明其新的 Windows NT 系统是可移植的（这也许吓了 Intel 公司一跳），而不是真的想打破他们在 PC 市场的联手垄断。对于 MIPS，结果并不那么好，芯片的销量不足以支撑公司盈利，其系统业务开始下滑。不久问题变得严重到连该公司的未来能否生存都成了问题。

SGI 收购 MIPS

进入 1992 年后，曾经希望能有基于 MIPS 处理器的符合 ACE 标准的新系统大量出现，已经被证明是遥不可及的事。偏偏就在此时，DEC ——作为 MIPS 的影响力最大的工作站用户——决定下一代的系统将转而采用自己的 Alpha 系列处理器。

这就使剩下的工作站公司 SGI，成了采用 MIPS 处理器的最大用户。到了 1993 年初，局势已经很明朗，只好由 SGI 出手挽救 MIPS 公司，因为它自己的生意依赖于 MIPS 体系结构，要以此举保证该体系结构未来不会消亡。到了 1994 年底，SGI 推出的新型号的 R4400 CPU（加大了高速缓存和经过了性能优化的 R4000）运行时钟达 200–250 MHz，让 SGI 稳居 RISC 性能的领先者行列。

1.4.4 QED: 嵌入式系统里的快速 MIPS 处理器

一些 MIPS 公司的关键设计人员离开后创建了一个新的名叫量子效应设计 QED（Quantum Effect Design）的公司。QED 的创始人全程参与过从 R2000 到

R4000 的 MIPS 处理器的设计。

有 IDT 作为其制造商和早期的投资者，QED 的小团队开始创建一个简单快速的 64 位 MIPS 的实现。该计划是想创建一个能够在合理的售价上提供良好性能的处理器，以便在许多应用中都能找到一个立足之处，包括低端工作站、小型服务器，直到象高级的激光打印机和网络路由器这样的嵌入式系统。

有些人坚决主张把 R4000 应用于嵌入式系统，但是阻力很大。QED 保证说 R4600 对嵌入式系统设计者的吸引力要比 R4000 大得多。R4600 很快就取得了成功。它重新回到简单的五级流水线设计并在合理的价位上提供了富有竞争力的性能。在 Cisco 路由器和 SGI 的 Indy 桌面系统中赢得了一席之地后，R4600 成了另外一个第一：第一个明显盈利的 RISC CPU。

QED 的设计团队继续改进他们的工作，在 1990 年代中期推出了 R4650 和 R4700。在讲到 R5000 时，我们回头再展开 QED 的故事。

1.4.5 R10000 处理器及其后继者

在 1990 年代中期，SGI 对高端工作站和超级计算机非常重视。纯粹的性能成了一个重要的卖点，其 MIPS 部门被要求在下一代的处理器设计中应对这一挑战。

SGI/MIPS R10000 于 1996 年早期发布。它是 MIPS 从传统的简单流水线的重大偏离；它是第一个真正利用了乱序执行和多指令发射的 CPU 。在几年之内，乱序执行设计横扫了所有的 CPU 设计，所有真正高端的现代 CPU 都是乱序执行的。但是当时验证和调试 R10000 的极度困难，使得参与者和旁观者都认为 SGI 采用这样激进的设计是一个错误。

SGI 的工作站业务在 1990 年代后半期开始开始亏损，不可避免的削弱了其对 MIPS 体系结构继续投资的能力。正好在这一时候，主流 PC 市场继续蓬勃扩张，导致了源源不断的资金收入流进来，为与 MIPS 竞争的体系结构——最为瞩目的就是 Intel 公司的奔腾系列及其后继的产品，在一定的意义上也包括 IBM 和 Motorola 的 PowerPC ——的未来发展提供了充足的资金。

在这种大背景下，SGI 开始研发 R10000 之后的 MIPS CPU；但是由于面临不断增加的资金压力，该项目在设计组未能完成之前就取消了。1998 年，SGI 公开承诺在其未来的工作站中将采用 Intel 的 IA-64 体系结构，最后一个桌面/服务器产品的 MIPS 开发团队解散了。2006 年（在我写作本书时）一些 SGI 的机器仍然在用 R16000 CPU ；该 CPU 尽管利用工艺的进步达到了更高的时钟频率，但其内部设计与 1996 年的 R10000 相比只有极少的增强。同时，IA-64 CPU 的销量远远低于 Intel 的最悲观的估计，世界上最快的 CPU 都是 x86 的变体。SGI 在选择 CPU 的时候，真的是倒霉到家了。

1.4.6 消费类电子产品领域的 MIPS 处理器

LSI Logic 和 Sony PlayStion

1993年，Sony 和 LSI Logic 签约开发用作第一代 PlayStation 的芯片。该芯片基于 LSI 的 CW33000 处理器核，时钟频率为 33 MHz 并且集成了 DRAM 控制器和 DMA 引擎等几个外围功能。PlayStaion 高度集成化的设计降低了生产成本，加上其前所未有的 CPU 处理能力造就了很好的游戏机。Sony 很快超过了那些老牌厂商，成为游戏控制台市场的老大。

Nintendo64 和 NEC 的 Vr4300 处理器

Nintendo 游戏机输给了 Sony 许多市场份额。作为反击，Nintendo 联合了 SGI 决定跳过 32 位微处理器，在售价仅仅 199 美元的游戏机上直接采用 64 位芯片。

处于心脏地位的芯片——NEC Vr4300 ——就是一个裁减过的R4000，但是裁减得不多。为减少针脚数目以降低封装成本，该芯片确实只有 32 位的外部总线，而且在整数和浮点运算间共享逻辑部件。但是相对于 199 美元的售价，功能已经很强大了。

Vr4300 的计算能力、低价格、低功耗节电使得它在别处也非常成功，特别是在激光打印机领域，这为 MIPS 体系结构在“嵌入”式应用中又找到了一块用武之地。

但是 Vr4300 成了最后一个冲击游戏市场的通用处理器；到 1990 年代后期，这个市场的 CPU 设计变得越来越专业化，紧紧地和专用的硬件加速器结合在一起以实现 3D 绘制、纹理映射以及视频回放。当 Sony 重返 PlayStation 2 的时候，已经采用了一个性能优异的 64 位的 MIPS CPU 作为核心芯片。该芯片由 Toshiba 制造，拥有一个浮点协处理器，其吞吐量不输于任何一款 1988 年的超级计算机。事实该芯片过于专业化了以至于在游戏市场之外不能找到任何应用。同一 CPU 的另一个版本用在 Sony 的 PSP 手持游戏台上，在未来几年内应该在市场可以见到。

这些游戏机在全世界的累计销量超过几千万，占据了 MIPS CPU 处理器销量的最大一块，比其它任何领域卖出的 MIPS 处理器都多——也比许多别的 CPU 体系结构卖出的数量多。

1.4.7 网络路由器和激光打印机领域的 MIPS

R5000 处理器

R4600 及其后继产品的成功之后，QED 的下一个主要设计就是 R5000。于 1995 年推出——同年 SGI 推出了 R10000 ——也是一个超标量的实现。但是从通常的设计理念和复杂度上来说，这两个设计大相径庭。

R5000 采用了经典的五级流水线并且按顺序发送指令。但也能够同时发送一条整数指令和一条浮点指令。MIPS 体系结构使得这一策略实现相对简单；两个指令各自用一套寄存器和执行单元，所以识别双发送机会的逻辑并不需要多么复杂。

当然了事情的另一面就是性能增加相对有限。除非 R5000 被用在一个运行大量浮点运算的系统，否则超标量能力根本用不到。尽管如此，R5000 集成的其它改进使得系统设计者容易从 R4600 升级。

QED 成为一个没有工厂的半导体厂商

在其一开始的头几年，QED 作为一个纯粹的出售知识产权的公司运行，向半导体公司发放自己的处理器设计授权让他们生产和销售芯片。到了 1996 年，该公司觉得以自己的名义出售芯片可能会更好些。生产仍然由外部的合作伙伴进行——该公司自己仍然不生产（即它并不直接拥有任何一家工厂）——但是现在 QED 负责芯片的测试并自己处理所有的销售、市场以及技术支持。

在这段时间前后，QED 启动了一个开发 PowerPC 实现的项目，采用和 R4600 同样简约高效的风格。不幸的是，和潜在意向客户业务签约方面遇到的困难，让他们冲昏了头脑，结果导致该产品一直都没有进入市场。在这次向 PowerPC 简短的进军碰壁之后，QED 重新回来埋头于 MIPS 体系结构。

QED 的 RM5200 和 RM7000 处理器

QED 的第一个“自有品牌” CPU 是 RM5200 系列，R5000 的直接后续产品。拥有 64 位外部总线的产品在网络路由器中工作很好，其 32 位总线的低成本封装很适合于激光打印机。

QED 在 RM5200 成功的基础上于 1998 年发布了 RM7000。这款产品在 MIPS 的实现上刻下了几个重要的里程碑：是第一个引入（256K）片上二级缓存[5]。RM7000 还是一个真正够格的的超标量设计，除了从 R5000 继承下来的整数/浮点指令组合外，还能够同时发送多对整数指令。

RM5200 和 RM7000 在 1990 年代中后期一直销售良好，卖到了许多高端的嵌入式应用中，特别是在网络路由器和激光打印机中得到了广泛应用。QED 明智的确保 RM7000 在编程和系统设计两方面都能充分兼容 RM5200。这使得通过简单地升级到 RM7000 就能让日过中天的 RM5200 系统中年得志而重新焕发活力，许多客户发现沿袭这一路线对他们有利。

SandCraft

1998 年前后，为 Nintendo 设计 Vr4300 的团队成立了一家公司 SandCraft，着手生产被 QED 的 RM5200 和 RM7000 家族占据市场的高端嵌入式 CPU 。

[5]QED 起先希望用一个类似 DRAM 的存储器使得 RM7000 的二级缓存能做得很小，但结果发现在单个芯片上要兼顾 DRAM 高速度和高密度在当时是不可能实现的。到现在也还不能。

SandCraft 体系结构的设计目标很高，很是花了一些时间才进入市场。尽管该公司经过几年的努力建立了一个足够大的客户群，但是最终还是倒闭了，其资产被 Raza Technologies 所购得。现在还不清楚 SandCraft 留下的技术遗产是否有一部分最后能够设法投产而重见天日。

1.4.8 当今时代的 MIPS 处理器

Alchemy Semicondutor：低功耗 MIPS 处理器

到 1999 年时，移动电话、个人商务通和数码相机的市场一直在飞速增长。这类处理器优先要考虑的是低功耗：因为这些设备小巧轻薄，要靠电池供电，必须设计成能在极低的功率下工作。同时，竞争压力要求每一代设备要比其前代提供更多的特性。制造厂商在寻求 32 位计算能力来满足日益增长的需求。

这些因素结合到一起，给处理器设计提出了一个动态的目标：在（随着电池化学和制造技术的进步）缓慢增长的给定功率限制下，从每一代处理器设计中交付明显的性能提升。

也许这真的就是历史的巧合，此前没有人去实现一个快速低功耗的 MIPS 处理器。但 DEC 已经制造了一个 200 MHz 的低功率 ARM（“StrongARM”），ARM 公司当时也愿意构建一些不太高档的机器，以便让电池的寿命支撑更长的时间。当时 DEC 公司正在不明智地与 Intel 公司打专利官司，他们输了，输掉了官司，还输掉了大量宝贵的时间。Intel 从官司中赢得的一个收益就是可以开发 StrongARM 。奇怪的是，Intel 似乎花了两年时间才留意到这块宝贝的价值，但到那时所有的开发者都已经走光了。

1999 年成立的 Alchemy 公司就是为了抓住这个机会。借助于 Cadence ，一个芯片设计工具开发商的支持，一些 StrongARM 设计团队的成员转向了设计超低功率的 32 位 MIPS CPU。产品运行得很好，但是其设计过于高端，而且可能也有些晚了，以致于已经无法撼动 ARM 在移动电话中先入为主的地位。

Alchemy 转而把希望寄托于个人数字助理的市场，其当然需要比移动电话更快的 CPU。但是该市场并没有同样增长。而且，这个市场市场似乎成了一切创新都赔钱的市场；最终，微软对在 MIPS 上支持 Windows CE 先热后冷的态度使得 MIPS 体系结构在这一领域成为跛脚。

SiByte

这个公司也成立于 1999 年，以一个富有经验的设计团队为中心，也包括一些原来的 DEC Alpha 和 StrongARM 项目[6]的一些成员。

SiByte 构建了一个高性能的 MIPS CPU 设计——它的目标是在 1 GHz下实现高效的双发送。而且将采用容易集成到许多芯片级的产品中去的封装；有些

[6]在美国市场，一个取消的项目也可以产生不亚于一个成功项目的创新成果。

变种强调计算能力，包含多个 CPU 核，另一些变种把重点放到集成一些控制器以提供灵活的接口。

Sibyte 的设计在已经采用 QED 的 RM5200 和 RM7000 的网络设备制造商中间引起了很大的兴趣；然而当该公司投产的时候，制上造的困难导致了很久的拖延，这也证明实现 1 GHz 的目标十分困难。

公司购并：PMC-Sierrra、Broadcom、AMD

1990 年代的最后几年经历了臭名昭著的“dotcom 泡沫”。许多小的技术公司纷纷上市，然后其股票价格一路攀升，在几个星期内就上涨到了令人头晕目眩的高度。

网路公司是股票市场的宠儿，其市值狂升到几百亿美元，这时他们发现收购提供有用技术——这有时就包括 MIPS CPU ——的公司其实很容易。

就是在这种气候下，Broadcom 公司收购了 SiByte，PMC-Sierra 收购了 QED ——都是在 2000 年中。看上去好像高端 MIPS 设计在嵌入式系统中的未来，由于这两个新的母公司收购而上了双保险。然而好景不长！

技术股泡沫的破裂来势极为迅猛。到 2001 年后半年，网络公司的股票变得极其低迷，来自客户的订单已跌破警戒线；到了 2002 年整个行业发现自己处在剧烈的下滑之中。有些公司的市值在不到一年之内跌破了最高值的十分之一。

随之而来的跌势不可避免的影响到 PMC-Sierra 和 Broadcom 是否有能力继续执行他们原来对 QED 和 SiByte 处理器设计的计划。这时已经不光是钱的问题；随着许多成熟的产品线的销售额锐减到简直就如同屋檐上滴下的水滴那样时断时续，这些公司甚至连找一个到适当的战略方向也变得极度困难。

Alchemy Semiconductor 也感受到了剧变的寒风。尽管其设计的功耗低得令人动心，该公司也难以找到一个大客户。后来到了 2002 年，Alchemy 被 AMD 所收购，此后继续销售了几年其 Au1000 线产品。当本书付印的时候，我们听说 Alchemy 的产品线已经被 Raza Technologies 收购。

高度集成化的微处理器设备

Broadcom 一开始宣布了其计划将 SiByte 的 1250 设计从双 CPU 核演化为四核，同时集成额外的内存控制器和快的多得接口。这个项目成了整个行业下滑的受害者，1250 产品线的发展前途落入风雨飘摇之中。

同一时刻，QED 的设计团队——现在运作为 PMC-Sierra 的 MIPS 处理器部门——推出了自己的双 CPU 产品 RM9000x2。这也集成了一个 SDRAM 控制器和各种其它接口。部分由于整个市场的严酷状况，RM9000 尽管确实跨越了 CPU 时钟频率 1 GHz 的里程碑关口，其寻找客户的过程仍然极为缓慢。随后的衍生产品加入了更多的接口，包括以太网控制器，但是该设计要获得高回报依然存在困难。

2006 年，这类高度集成的设备的前景看上去令人疑虑。随着芯片中的晶体管不断缩小，从同样的生产成本所获得逻辑门数增加了。但是将新设计投产的一次性成本逐渐随之增加：对最近的 90 纳米工艺，这个成本超过一百万美元。如果你把可用的硅片面积都填上逻辑以增强芯片的能力，设计和测试的成本将以千万美元来计。

为了收回成本，每种芯片产品生命周期内需要卖出几百万片。象 RM9000x2 或者 Broadcom 的 1250 的每一个版本只能卖出几万或者几十万：这个数量根本不够。现在还不清楚未来哪种类型的设备能够吸引足够的批量从而为全定制的嵌入式 CPU 设计提供资金。

Intrinsity: 将 MIPS 处理器推进到 2 GHz

细心的读者可能已经注意到，截止目前 MIPS 发展的呈弧形曲线：早期的 R2000/R3000 实现曾是微处理器性能的领先者，但是竞争者的产品最终赶了上来并超过了 MIPS。

所以，也许你会问：在过去的几年中有没有人试图制造一个真正快速的 MIPS CPU 呢？答案是有的：Instrinsity Semiconductor 于 2002 年发布了其 FastMath 处理器。通过仔细的设计技术，从基本上是标准的工艺技术中挖掘出了更高的性能，Instrinsity 能够制造出一个精干的 32 位 MIPS 的设计，其主频达到令人瞩目的 2 GHz。

尽管这被广泛视为一项杰出的技术成就，该产品却一直在为寻找市场而苦苦挣扎。它还远没有现代的 PC 处理器那么快，功耗和散热问题从消费者的标准来看也有点高。

MIPS 公司的复活

1998 年，SGI ——面临日益增长的资金流问题——决定分离出其 CPU 设计部门，重新成为独立的 MIPS Technologies 公司。该新公司接到一个合同，要创建一个作为 SoC 的一部分的核心 CPU 。你可能还记得我们很早以前在本节碰到过 SoC 的想法，那是在我们描述第一个 ASIC 核出现的时候。

在 SoC 的早期的日子里，CPU 厂商发现很难保证一个核的性能——例如，CPU 时钟频率——除非他们对预定义的每一个可能的芯片“代工厂(foundry)”制造工艺向客户提供一个核的内部结构的固定的物理版图——“硬核”。

MIPS 本来想构建高端的硬核以及生产和供应快速的 64 位设计（20 Kc 和后来的 25 Kf）。但是他们赌错了。最近几年中，市场只对它的可综合的核（起先叫做“软核”）青睐有加。

一个可综合的核是一系列用来描述电路的可以编译成真正的硬件设计的（通常为 Verilog）设计文件。一个可综合的核产品不止是 Verilog 设计，还要包括很多其它东西，因为客户必须能够把核加入到一个更大的 SoC 设计中去，并且验证 CPU 及其连接完好，保证整个芯片可以工作。

MIPS 公司的第一个可综合的核是中档的 32 位 4-K 系列；从那以后，又加入了 64 位的5K，高性能的的 32 位 24 K，以及（2006 年初发布的）多线程的 34 K。

1.4.9 今天

今天还在用的 MIPS CPU 主要在以下四大类：

- *SoC 核*:MIPS CPU 至今在尺度和功耗方面受益于斯坦福项目的简单设计。一个拥有长期历史，遍及从手持设备到超级计算机的体系结构相对于专门为低端定做的体系结构而言是一个极具吸引力的替代方案。MIPS 是第一个可用于 ASIC 核的“长大成人”的 CPU ——目睹了自己在 Sony PlayStation 游戏机控制台上的存在。最突出的供应商的就是 MIPS 公司，但是 Philips 也有自己的设计。

- *集成的嵌入式 32 位 CPU*: 从几个美元起价，这些芯片包含 CPU、缓存和基本的面向应用的模块（网络控制器很受流行，受欢迎），各自价格、功耗、处理能力都差别很大。尽管 AMD/Alchemy 有些极具吸引力的产品，这个市场似乎是命中注定了厄运。在该目标市场的 CPU 芯片，发现一个 SoC 在最大化集成度方面做得更好，节省下了关键的成本和功耗。

- *集成的嵌入式 64 位 CPU:* 这些芯片对于高端嵌入式应用提供了极具吸引力的速度/功耗折中。网络路由器和激光打印机时是这一类中最常见的应用。但是看上去很难有足够的销售量能保证支付芯片开发的成本。

  但是这一类公司中有一些正在尝试激进的新设计思想，它们给 MIPS 体系结构贡献了许多清新的概念，常常是做真正前沿探索的最适合的出发点。Raza 的 XLR 系列的多核、多线程处理器代表了另外一种类型的嵌入式 CPU，其目标是比“传统”的嵌入式 CPU 提供更多的附加值（单个芯片带来更多的收益）。Cavium 公司的 Octium 也很有前景。

- *服务器处理器:* SGI，曾经收养了 MIPS 体系结构的的工作站公司，继续供应高端的 MIPS 系统，一直到 2006 年破产为止，尽管此时距其承诺未来采用 Intel IA-64 已经七年了。但是这些高端系统的路已经走到了尽头：MIPS 注定“只能”局限在广阔的消费者和嵌入式市场了。

表 1.1 列出了一些里程碑意义的产品的主要特性。我们还没有讨论从 MIPS I 到 MIPS64 的指令集的版本变迁。这些内容将在2.7 节详细讨论，你也可以从该节得知 MIPS II 的命运。

表 1.1: MIPS CPU 的里程碑

年份	厂商/型号/主频(MHz)	指令集	高速缓存（I+D）	备注
1987	MIPS R2000-16	MIPS I	片外4K+4K~32K+32K	片外（R2010）浮点单元 FPU
1990	MIPS R3051-16		4K+1K	第一个具有片上缓存的 MIPS CPU，同时也是一族在针脚级兼容产品的始祖
1991	MIPS R4000-100	MIPS III	8K+8K	集成浮点部件和二级缓存控制器，可选针脚数。完全的 64 位 CPU——但是五年后还很少有 MIPS CPU 真正利用其 64 位指令集。长流水线和半速接口有助于提高时钟频率。
1993	IDT/QED R4600-100		16K+16K	经 QED 精心调整后的设计比同时钟的 R4000 和 R4400 快得多——部分由于重新回到了经典的五级流水线。是 SGI 的快速低端 Indy 工作站和 Cisco 的路由器的重要部件。
1995	NEC/MIPS Vr4300-133		16K+8K	R4000 低成本、低功耗但全功能的衍生产品。一开始时针对 Nintendo 64 位游戏机控制台，其嵌入式应用包括惠普的 LJ4000 激光打印机。
1996	MIPS R10000-200	MIPS IV	32K+32K	R10000 非比寻常，其中充满了微处理器技术的创新。它坚持的主要 MIPS 传统是按一个原则做到极致。结果是发热大、不好用，但是每兆赫性能无可比拟。
1998	QED R7000-		16K+16K+256K L2	第一个拥有片上二级高速缓存的 MIPS 处理器，成为几代激光打印机和因特网路由器的核心。
2000	MIPS 4 K 核系列	MIPS32	16K+16K（典型值）	到目前为止最成功的 MIPS 核——可综合而且成本低。
2001	Alchemy Au-1000		16K+16K	如果想要 500 mW 功率能达到 400 MHz 速度，这将是你的不二之选。但该产品在市场上并不成功。
2001	Broadcom BCM1250	MIPS64	32K+32K+256K L2	600 MHz 的双 CPU 设计（共享二级高速缓存）。
2002	PMC-Sierra RM9000x2	MIPS64	16K+16K+256K L2	1 GHz 的双 CPU 设计（每个 CPU 各自拥有 256 K 非共享的二级高速缓存）。第一个达到 1 GHz 的 MIPS CPU。
2003	Intrinsity FastMath	MIPS32	16K+16K+1M L2	具有向量 DSP，主频高达 2 GHz 的 CPU，但是市场却对其敬而远之。
2003	MIPS 24K 核	MIPS32 R2	16K+16K 32K+32K 64K+64K	综合后的核能达到 500 MHz，该核的设计极为成功。
2005	MIPS 34K 核	MIPS32+MT ASE	32K+32K（典型配置）	MIPS 多线程的先驱者。

1.5 MIPS 和 CISC 体系结构比较

具有早期体系结构汇编语言知识的程序员——特别是那些在 x86 或者 680x0 CISC 指令集下成长起来程序员——可能会对 MIPS 指令集和寄存器模型感到奇怪。我们在这里尽量做一个总结，这样你就不至于会以后脱离正轨去寻找一些 MIPS 中根本不存在的东西，比如堆栈操作的 push/pop 指令。

我们将考虑以下几方面：为提高流水线效率而给 MIPS 操作加上的限制；极端简单的 load/store 操作；可能是有意省略的操作；指令集的一些预想不到的特性；流水线操作中对程序员可见的地方。

最初提出 MIPS 设想的斯坦福小组，特别关注实现简短的流水线。后来的事实进一步证明了他们的判断：由流水线而引出的许多决定也适合于更容易和更快速地实现更高的性能。

1.5.1 MIPS 指令集的限制

- *所有指令长度都是 32 位：* 这意味着没有指令能够仅占用两三个字节的内存空间（因而 MIPS 的二进制文件比典型的 680x0 或 80x86 大 百分之二十到三十），也没有指令可以超过四个字节。

  随之而来就是不可能把一个 32 位常数放进单个指令中（指令中没有足够的位用来编码操作数和目标寄存器）。MIPS 设计者决定留出 26 位常数的空间用以编码跳转和调用指令的目标地址；但是仅有给两条指令。其它指令只能有 16 位空间留给常数。这样装入任意 32 位数值需要一个两条指令的序列，条件分支被限制到 64 K 指令范围。

- *指令操作必须适合流水线：* 只能在相应的流水线阶段才能执行任务，并且必须在一个时钟周期内完成。例如，寄存器写回阶段只能有一个值存入寄存器堆，所以指令只能修改一个寄存器。

  整数乘除是不可或缺的重要指令，但是不能在一个时钟周期内完成。MIPS CPU 传统的做法是发送这些操作到一个独立的流水线单元，这个我们以后再说。

- *三操作数的指令：* 算数/逻辑指令不需要指定内存地址，所以空出了充足的指令位可以定义两个独立的源操作数和一个目的操作数。编译器喜欢三操作数指令，其给了优化程序更大的空间来优化处理复杂表达式的代码。

- *32 个寄存器：* 寄存器数量的选择主要是由软件需求驱动的，在现代体系结构中一组 32 个通用寄存器是最为流行的。采用 16 个肯定不够现代编译器的需要，但是 32 个足够让 C 编译器把（除了最大且最复杂的函数之外）常用的数据保存在寄存器中。采用 64 个或更多的寄存器需要更大的指令域去编码寄存器而且也增加了上下文切换的负担。

- *寄存器零:* $0 寄存器永远返回零，给这个常用的数提供一个简缩的编码。

- *没有条件码:* MIPS 的指令集的一个特征就是没有条件标志，这即使在 1985 年的 RISC 中也是极为激进的。许多体系结构有多个标志位来表示运算结果的“进位”、“为零”等等。CISC 的典型做法是根据一些指令的操作结果设置这些标志，有些 RISC 体系结构保留了标志位（尽管往往只能由比较指令明确设置）。

  MIPS 体系结构决定把所有信息保存到寄存器堆中。比较指令设置通用寄存器，条件分支指令检测通用寄存器。那样确实有利于流水线实现，因为能够减少对算术/逻辑操作依赖的巧妙机制不论哪一种也都同时会减少比较/分支指令对中的依赖。

  我们后面会看到有效的条件分支意味着是否分支的决定必须在半个流水线周期内作出；该体系结构通过保持分支决策的测试条件简单有助于实现这一点。所以 MIPS 的条件分支只测试单个寄存器的符号/为零或者一对寄存器是否相等。

1.5.2 寻址和访存

- *访问内存只能通过简单的寄存器加载和存储:* 对内存变量进行算术运算会打乱流水线，所以不这么做。每次内存访问都都要一条显式的加载或存储指令。大的寄存器堆使得这一点实际远没有听上去那么麻烦。

- *只有一种数据寻址方式:* 几乎所有的加载和存储都通过单个寄存器基址加上一个 16 位的常数偏移量（浮点指令可以使用有限的寄存器加寄存器）寻址内存。

- *字节地址指令:* 一旦数据存入 MIPS CPU 的寄存器，所有的操作都是在整个寄存器上操作。但是象 C 这样的语言语义不适合不能寻址内存到字节粒度的机器。因而 MIPS 对 8- 和 16- 位变量（分别称为字节和半字）提供了一套完整的装入/存储操作。一旦数据到达寄存器，就当作寄存器全长来处理，所以部分字加载指令有两种形式——符号扩展和零扩展。

- *load/store 必须对齐*: 内存操作只能从对齐到相应数据类型边界的地址加载和存储数据。字节可以在任意地址传输，但是半字必须在偶数地址对齐，字在四字节边界对齐。许多 CISC 微处理器可以从任意字节地址加载/存储四字节数据，但是要花费额外的时钟周期。

  但是，MIPS 指令集体系结构（ISA）确实包含有几个特殊的指令以简化对没有适当对齐的地址存取操作。

- *跳转指令:* 有限的 32 位指令长度在想要支持很大程序的体系结构上对分支是个问题。MIPS 指令的最小操作码域为 6 位，留出了 26 位来定义跳转

的目标。因为所有指令在内存中都是四字节边界对齐的，低两位地址无需保存，这样可有 $2^{28} = 256$ MB 的地址范围。这个地址不是相对 PC 的，而是解释成 256 MB 段内的绝对地址。这对大于 256 MB 的单个程序极为不便，但到目前还没有碰到太大的问题。

超出段内的分支可以通过使用一个寄存器跳转指令做到，该指令可以跳到任意 32 位地址。

条件分支只有 16 位的偏移域——给出了 2^{18} 字节的范围，因为指令都是四字节对齐的——解释成相对 PC 的带符号的偏移量。如果知道分支目标会在紧跟分支之后的指令的 128 KB 范围内，编译器就能只生成一个简单的条件分支指令。

1.5.3 MIPS 没有的特性

- *没有字节或半字数据的运算:* 所有算术和逻辑操作都在 32 位的数据上进行。字节或半字的运算需要大量额外的资源和许多额外的操作码，而且很少有用。C 语言的语法让大多数的计算用 int 类型，对 MIPS 而言 int 就是 32 位的整数。

  然而当程序明确做 short 或者 char 运算时，MIPS 编译器必须插入额外的代码以保证结果回绕和溢出，生成跟 16- 或 8- 位机器上一样的结果。

- *没有对堆栈的特殊支持:* 传统的 MIPS 汇编确实定义了一个寄存器作为堆栈指针，但是硬件上 SP 没有任何特殊之处。有一种推荐的关于子程序调用的栈帧布局，这样可以混合不同语言和编译器的模块；你应当遵守这些约定，但是这些与硬件无关。

  堆栈弹出不能适应流水线，因为有两个寄存器要写（来自堆栈的数据和增加指针值）。

- *最少的子程序支持:* 有一点比较特别：跳转指令有一个跳转并链接的选项，把返回地址存入一个寄存器，默认是 $31 。所以方便起见习惯上用 $31 作为返回地址寄存器。

  这样做比起把返回地址保存到堆栈上要简单，但却带来明显的好处。随便举两个好处瞧瞧：第一，保持了分支和访存指令的完全分离；第二，当调用许多根本不需要在堆栈保存返回地址的小程序时，这样做有助于提高效率。

- *最少的中断处理:* 很难看到硬件能做得比这更少的了。它把重新开始的地址存放到一个特殊的寄存器，接着仅修改刚刚够找出怎么回事的少量机器状态并禁止进一步中断，然后跳转到低端内存事先定义好的一个单一入口地址，此后一切由软件负责。

- 最少的异常处理：中断只是异常的一种类型（MIPS 中异常这个词包括所有需要 CPU 中断正常的顺序执行并调用软件处理的事件）。一个异常可以来自一个中断、来自对物理上不存在的虚拟内存的试图访问、或者其它很多情况。一条有意引入的、类似系统调用的、用来进入受保护的 OS 内核的自陷指令发生时，也会进入一个异常。所有异常都导致控制传递到同样的固定入口地址。[7]

  任何异常发生时，MIPS CPU 都不会存进堆栈、写入内存或者备份寄存器。

  按照约定，保留了两个通用寄存器给用于异常，这样异常处理程序可以自举（在 MIPS CPU 上没有寄存器不可能做任何事情）。对于运行在允许中断和自陷的任何系统上的程序来说，这两个寄存器的值随时可能变化，所以最好不要用。

1.5.4 程序员可见的流水线效果

到此为止，以上就是你需要从一个简化的 CPU 了解的全部内容。然而使得指令集适应流水线也会导致一些奇怪的效果。为了便于理解，我们画图来说明。

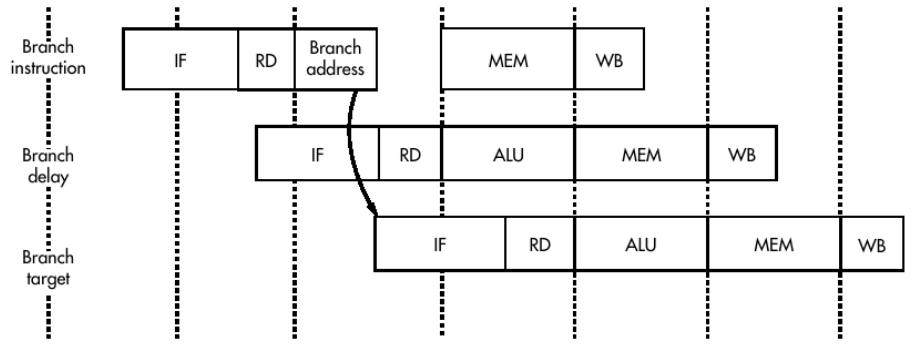

图 1.3: 流水线和分支延迟

- 延迟分支：MIPS CPU 的流水线结构（图 1.2）意味着当一个跳转/分支指令到达执行阶段产生新的程序计数器值时，跟在跳转指令后的指令已经开始了。该体系结构并不是丢弃这部分有潜在用途的工作，而是要求紧跟分支后的指令总是在分支目标指令之前执行。紧接分支指令之后的指令位置称为分支延迟槽。

  要是硬件没有特殊处理，是否分支的决定以及分支的目标地址，就会在 ALU 流水阶段结束时得到——到此时，如图 1.3 所示，已经太晚了，甚至

[7] 略有夸张；这年头实际上都有好几个不同的入口点，至少两个。详情请参阅 5.3 节。

在下下一个流水线槽都来不及提供一个指令地址。

但是分支指令的重要性足以给其特殊处理。从图 1.3 可见，提供了一条经 ALU 的特殊路径可以让分支目标地址提早半个周期到达。连同取指阶段多出来的半个时钟周期的偏移，就刚好来得及取出分支目标指令作为下下一个指令。这样硬件就会运行分支指令、接着运行分支延迟槽指令、然后是分支目标指令——再没有其它的延迟。

编译器系统或者汇编程序应该考虑甚至利用分支延迟；结果是通常有可能通过适当安排使得延迟槽中的指令做些有用的工作。经常可以把别处的指令移到延迟槽中。

对于条件分支问题会有点复杂，分支延迟指令（至少）应当对两条分支路径都无害。实在找不到有用的事情可做时，延迟槽中填入一条 nop 指令。

除非明确要求，否则许多 MIPS 汇编器都对程序员隐藏这个古怪的特性。

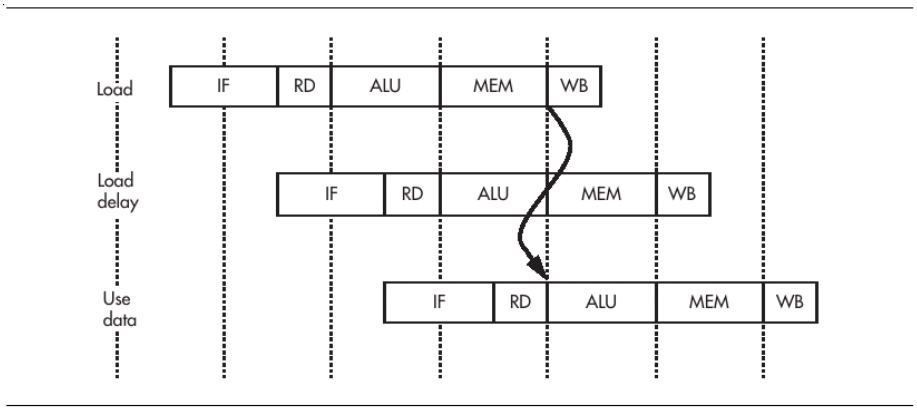

图 1.4: 流水线和加载延迟

- *数据加载延迟（加载延迟槽）*：流水线的另一个后果就是一条加载指令的数据在下一条指令的 ALU 阶段的开始才从高速缓存/内存系统到达——所以在下一条指令中不能使用加载的数据。（参看图 1.4 以理解其工作机制）

  紧接加载指令之后的指令位置称为 *加载延迟槽*，一个优化的编译器将试图用它做些有用的事情。汇编器对程序员隐藏这一点，但可能插入一条 nop 指令。

  在现代的 MIPS CPU 上，加载结果是互锁的：如果你试图过早使用结果，CPU 将停下来等待数据到达。但是早期的 MIPS CPU 没有互锁，在延迟槽中试图使用数据将导致无法预料的结果。

第 2 章　MIPS 体系结构

在计算领域中, *体系结构*这个相当大的字眼用来描述一个用于编程的抽象机器，而不是一个机器的具体实现。这一点区别非常有用——值得和在市场炒作中对这个词的滥用区分开来。读者有可能不熟悉抽象描述，但对其概念并不陌生。如果你开过手动档变速车，就会发现不管该车是前轮驱动还是后轮驱动，总是油门踏板在右，离合器踏板在左。在这里，尽管具体实现（哪个轮子驱动）不相同，体系结构（即哪个踏板在哪边）却有意保持相同。

当然，如果你是一个公路拉力车赛手，考虑的是能够在打滑的路面上高速行驶，这时前轮还是后轮驱动就很有关系了。计算机也是如此——如果你有某种极端的或者超常规的需求，实现的细节就对你很重要了。

一般而言，一个 CPU 的体系结构包括一组指令集加上一些寄存器的知识。*指令集* 与*体系结构*这两个术语基本上是同义词。在缩写 ISA(Instruction Set Architeture 指令集体系结构)中你可以看到这两个术语同时出现在一起。

目前，对 MIPS ISA 最好的定义就是由 MIPS 公司出版的 MIPS32 和 MIPS64 体系结构规范。MIPS32 是 MIPS64 中用于只有 32 位通用寄存器的 CPU 的子集。“MIPS32 和 MIPS64 体系结构规范”说起来太长，我们下面简称“MIPS32/64”。

大多数制造 MIPS CPU 的公司现在都兼容这些规范——那些不能严格兼容的公司也在尽量缩小差别。

在 MIPS32/64 发布之前，写过好几个 MIPS 体系结构的版本。但是这些老定义的正式适用仅仅限于高层软件所用的指令和资源——而把操作系统必需的 CPU 控制机制看作是具体实现相关的。这就意味着可移植的操作系统工作依赖于一个要维持 MIPS 体系结构未规定部分稳定的君子协议。最后的做法是每个体系结构的版本都自然的与一个“母实现”相关联。

MIPS I: 最早的 32 位处理器（R2000/3000）使用的指令集；所有出现过的 MIPS CPU 都能运行这些指令，所以仍然是 MIPS 世界一种通用的“世界语”。

MIPS II: 为从来就没有正式投产的 R6000 机器定义了些微的升级。但是有一种很类似 MIPS II 的东西在嵌入式市场上的 32 位 CPU 中广泛使用。MIPS II 是 MIPS32 的前身。

MIPS III: 由 R4000 引入的 64 位指令集。

MIPS IV: 在 MIPS III 上增加了几条有用的（多为浮点运算）指令，出现在两个具体实现（R10000 和 R5000 ）中。

MIPS V: 添加了一些奇怪的同时执行两条浮点操作的（"SIMD"）指令——但是从来没有制造过 MIPS V CPU 。其中大部分指令以"成对单精度"的方式作为一个可选部分重新出现在 MIPS 64 中。

MIPS32, MIPS64: 1998 年从 SGI 分离出来之后的 MIPS 科技公司制定的标准。该标准第一次包括了称为协处理器 0 的"CPU 控制"功能。MIPS32 是 MIPS II 的超集，而MIPS64（其中包括 64 位指令集）是 MIPS IV 的超集（还以可选的扩展的方式包含了 MIPS V 的大部分）。

大多数 1999 年之后设计的 MIPS CPU 兼容这些标准，所以我们本书中将采用 MIPS32/64 作为基础的体系结构。更确切地说，我们的基础是发表于 2003 年的 MIPS32/64 规范的第二版。此前的 MIPS32/64 CPU 就称为第一版。

该体系结构标准定义了原先的公司文档中所定义的全部内容；直到 MIPS V 之前，这些文档定义的规范，足以保证能够运行同样的 UNIX 应用程序，但不足以保证用了操作系统或底层特性的代码能够完全可移植。作为对比，如果一个 CPU 遵循 MIPS32/64 规范，那就应当能够运行一个可移植的操作系统。

（自 R3000 开始的）早期 MIPS CPU 的 CPU 控制指令和寄存器与 MIPS32 有所不同；这些已经很老了，本书不再详述。

有不少 MIPS CPU 的实现加入了一些新的指令和有意思的的特性，但不是总能容易找到软件或工具(特别是编译器)来充分利用这些与具体实现相关的特性。我们将会提到大多数我认为有意思的特性。

可以在两种不同的层次上来描述 MIPS 的体系结构。第一种描述(本章)是，在工作站上编写一个用户应用程序时在汇编语言的层次上所看到的东西。这也意味着 CPU 的正常工作的整个过程是可见的。

后面的各章，我们将介绍各个方面，包括隐藏在高层的操作系统后面的令人讨厌的全部细节——CPU 控制寄存器、中断、自陷、高速缓存操作和内存管理。至少我们会将这些任务分解成一些小片断来介绍。

CPU 在用户级上的兼容性常常比暴露所有特性时的兼容性好得多。已知的所有 MIPS CPU 都可以运行 MIPS I 的用户级代码。

指令集扩展的规范化——ASE

我们反复强调，RISC 并不意味着指令集就小。事实上 RISC 的简单性使得通过对三操作数的寄存器到寄存器的运算进行变形来扩展指令集变得过于容易，

容易得让人经不住诱惑。

当 MIPS 刚刚在嵌入式系统应用的时候，针对具体应用而增加的新指令如雨后春笋般的冒了出来。MIPS32/64 吸收了其中一些指令，同时也允许以 ASE（专用指令集扩展）形式提供了一些规范。ASE 是对 MIPS32/64 的可选的扩展，如果有的话，其存在以一种标准方式通过配置寄存器来标识。现在已经有相当多的扩展：

- MIPS16e:该扩展远在 MIPS32 和 MIPS 公司之前就有了。在 1990 年代最先由 LSI 创始，着眼于缩小 MIPS 的二进制代码的大小。 MIPS16 将用户级 MIPS 指令的一个子集编码成 16 位的操作码（还增加了几条特殊的指令）。这样的指令集明显“太小”，程序编译成 MIPS16 产生的指令数远多于编译成常规的 MIPS 指令；但是这种长度减半的指令会产生小的代码。 MIPS16 指令集被组织成对 MIPS32 的扩展，扩展后的版本就叫做 MIPS16e。这本书很少涉及 MIPS16e 的细节：这个题目太大了，即使在汇编层次也难以讲清楚，但在附录 B.1 对此讲了几句。

- MDMX: 这是由 SGI 支持的另外一种老式扩展，增加了一大批采用浮点（协处理器 1 ）寄存器的 SIMD 算术操作指令。因为每个操作在 64 位寄存器中存放的短整型数组的每个元素上并行处理，所以是 SIMD。这类运算许多是“饱和”型的；即当结果溢出时，目标操作数用可表示范围内最接近的值来代替。小整数的饱和 SIMD 操作能够加速各种音频视频的“流媒体“计算。这在称为 DSP(Digital Signal Processing) 的专用 CPU 中是广泛采用的特性。

  MDMX 类似于 Intel MMX 扩展的早期版本。但是 SGI 自己从来没有实现 MDMX ，Broadcom 的 CPU 似乎是仅存的实现。

- SmartMIPS: 加密运算被认为是智能卡市场的关键，该扩展针对提高加密性能而对体系结构作了小小的扩充，同时结合了其它几个与 CPU 控制系统安全和尺寸相关的修订。将来有一天 32 位的智能卡将成为一个巨大的市场，但是现在这一切还没有发生。

- MT: 在指令方面的扩充很小，但却有着巨大的意义：给 MIPS 核加上了硬件的多线程功能。首先出现在 MIPS 科技于 2005 年发布的 34-K 族处理器中。附录 A 对 MIPS MT 作了一个概括介绍。

- DSP: 这是一组为音频视频处理保留的指令集，拥有对小整数的饱和算术以及 SIMD 操作，类似于 MDMX ——但在实际中已经明显比 MDMX 更为有用。这也是在 2005 年新推出的，用在 MIPS 科技的 24-K 和 34-K 族处理器中。DSP ASE 公布的时机恰好能够让我们在附录 B.2 中加一个简要的描述。

MIPS32/64 规范中还有一些其它可选的部分，不适合看作主流的指令集扩充：

- 浮点：最老的是最好的。从最早期的日子开始浮点就一直是 MIPS 指令集的可选部分，但是被限制在“协处理器 1”编码中。本书对此作了详细地介绍。
- CP2：第二协处理器编码区保留给胆大的客户自由使用。但是因需要大量的设计和测试工作，很少有人去试过。这里也没什么好讲的。
- CorExtend：MIPS 科技尽最大努力推出的一个还算易用的用户可定义指令集。这个概念在 2002/2003 年间被大肆炒作，ARM 和 Tensilica 当时都声称支持。
- EJTAG：为了增强调试功能而加的可选系统，在 12.1 节有介绍。
- 单精度浮点对：对浮点单元的扩展，提供 SIMD 操作，每个指令同时操作两个单精度（32 位）浮点数值。
- MIPS-3D：通常总是和成对浮点相结合，增加了几条指令，对在三维屏幕绘图中大量使用的浮点矩阵运算实现加速。

2.1 MIPS 汇编语言的风格初探

汇编语言是 CPU 原始二进制指令的可供人书写（以及阅读）的版本。第 9 章专门讲述汇编语言。从来没有接触过汇编语言的读者可能会对本书部分内容时感到难以理解——但是现在开始学还为时不晚。

对于熟悉汇编语言但不熟悉 MIPS 的读者，下面是一些例子可以参考：

```
    /* this is a comment */

entrypoint:                 # that's a label
    addu $1, $2, $3         # (registers) $1  $2 + $3
```

与大多数汇编语言一样，MIPS 汇编语言也是以行为单位的。一行结尾表示指令的结束，汇编语言注释约定忽略一行中“#”字符之后的内容。在一行里可以有多条指令，之间要用分号隔开。

单词后面跟着一个冒号“：”代表一个标号——这里的单词指广义的，可以包含各种奇怪的字符。标号用来定义代码中的入口点和命名数据区的一个存储位置。

MIPS 汇编器解释一个非常严格的、充满了寄存器编号的语言。大多数程序员用 C 预处理程序和一些标准头文件，这样可以用名字书写寄存器；通用寄存

器的名字反映了其习惯用法（我们将在第 2.2 节讲到）。如果你用了 C 的预处理程序，也可以使用 C 风格的注释。

许多指令都是如上所示的三操作数形式。目标寄存器在左边(注意，这一点与 Intel x86 习惯正相反)。一般而言，存放结果和操作数的寄存器的书写顺序与 C 语言或其他代数语言的方式是一致的，例如：

```
subu $1, $2, $3
```

意味着：

```
$1 = $2 - $3;
```

目前讲这么多就够了。

2.2 寄存器

程序可以用的通用寄存器有 32 个：$0 ——$31。其中有两个，也只有这两个的用法与其它的不同。

$0 不管存入什么值，永远返回零。

$31 永远由正常函数调用指令 (jal) 存放返回地址。请注意寄存器调用版本的指令(jalr)可以用任意寄存器来存放其返回地址，但是使用 $31 之外的寄存器有违常规。

其它方面，所有的寄存器都是一样的，可以在任何一个指令中使用(你甚至可以用 $0 作为一个指令的目标寄存器，但是这样指令的结果就无声的消失了。)

MIPS 体系结构中，程序计数器并不是一个寄存器，你最好想都不要那样想——在一个具有流水线的 CPU 中，程序计数器有多个可能的取值令人混淆。jal 指令的返回地址是其后的下下一条指令：

```
...
jal printf
move $4, $6
xxx # return here after call
```

这样做是有道理的，因为紧跟调用指令后面下一条指令是调用指令的延迟槽——请记住，按照规定该指令必须在分支目标指令之前执行。调用的延迟槽指令很少被浪费，因为经常可以用来建立调用的参数。

MIPS 里没有状态码。状态寄存器或 CPU 的其它内部状态对用户级程序没有任何影响。

有两个寄存器大小的结果端口（叫做 hi 和 lo ）与整数乘法运算相关。它们不是通用寄存器，除了乘除法指令之外没有其它用途。但是，定义了向这两个端口插入任意值的指令——仔细想一想你就会发现，这对于恢复一个被中断的程序的状态是必需的。

浮点数学协处理器(浮点加速器 FPA 或 FPU)，如果有的话，增加了 32 个浮点寄存器；简单的汇编程序中就称为 $f0–$31。

实际上，对于早期的(符合 MIPS I 标准的）32 位机器，只有 16 个偶数编号的寄存器可以用做数学计算。当然每个偶数号的寄存器既可以用一个单精度数(32 位)，又可以用一个双精度数(64位)；当你做一个双精度的运算时，寄存器 $f1 存放 $f0 的多余的位，以此类推。在 MIPS I 程序中，只有当你在整数和浮点寄存器之间传送数据或者加载/存储浮点寄存器值的时候，你才能看见奇数编号的寄存器——即便此时汇编器也帮你不用记住这些复杂性。

MIPS32/64 CPU 有 32 个真正的 FP 寄存器。但是还可能碰到一些软件，为了保持与过去的 CPU 兼容性而避免用奇数号的寄存器。在控制寄存器中有一个模式位可以让现代的 CPU 模拟老的行为。

2.2.1 通用寄存器的习惯命名和用法

我们已经花了几页描述了一些体系结构方面的内容，下面来介绍一些软件方面的内容。我觉得你现在有必要了解这些。

尽管硬件几乎没有使用寄存器的规定，但在实际使用寄存器时还要遵循一系列习惯约束。硬件根本不关心这些习惯用法，但是如果你想用别人的子程序、编译器或操作系统，那你最好遵守这些惯例。

与这些习惯用法相应的有一套习惯命名。如果要符合习惯用法，那就几乎不得不使用这些习惯命名。表 2.1 列出了常见的命名。

在 1996 年前后，SGI 引入了使用新的命名约定的编译器。新的约定可以用来生成使用 32 位或 64 位寻址的程序，这两种情形分别称为“n32”和“n64”。我们暂时不讨论这些，第 11.2.8 节对此作了一个简明但是相当全面的描述。

汇编程序的寄存器的习惯命名和用法

- at: 这个寄存器为汇编器产生的合成指令保留。如果你要显式的使用这个寄存器（比如在异常处理程序中保存和恢复寄存器）时，有一条汇编伪指令可用来禁止汇编器使用它——但是这样一来汇编器的一些宏指令就不能再用了。

- v0, v1: 用来存放子程序返回的的非浮点值。如果要返回的值太大，这两个寄存器放不下，编译器将会通过内存来完成。细节可参见 11.2.1 节。

- a0–a3: 用来传递子程序调用时前 4 个非浮点参数。不过偶尔情况可能要更复杂些。请参见11.2.1节。

- t0–t9: 依照约定，一个子程序可以不必保存而自由使用这些寄存器。这使得这些寄存器很适合用作表达式求值时的临时变量——但是编译器/程序员必须记住，这些寄存器中的值有可能被子程序调用破坏掉。

表 2.1: 通用寄存器的习惯命名和用法

寄存器编号	助记符	用法
0	zero	永远返回 0
1	at	(assembly temporary 汇编暂存)保留给汇编器使用
2-3	v0, v1	子程序返回值
4-7	a0-a3	(arguments) 子程序调用的前几个参数
8-15	t0-t7	(temporaries) 临时变量，子程序使用时无需保存
24-25	t8-t9	
16-23	s0-s7	子程序寄存器变量；子程序写入时必须保存其值并在返回前恢复原值，从而调用函数看到这些寄存器的值没有变化。
26,27	k0,k1	保留给中断或自陷处理程序使用；其值可能在你眼皮底下改变
28	gp	(global pointer)全局指针；一些运行系统维护这个指针以便于存取 `static` 和 `extern` 变量。
29	sp	(stack pointer)堆栈指针
30	s8/fp	第九个寄存器变量；需要的子程序可以用来做帧指针(frame pointer)
31	ra	子程序的返回地址

- s0–s8: 依照约定，必须保证这些寄存器的内容在子程序返回时要和子程序入口时的值相同，要么在子程序里不用这些寄存器，要么把它们保存在堆栈上并在子程序返回时恢复。这种约定使得这些寄存器非常适合作为寄存器变量，或存放一些在子程序调用期间必须保存的值。

- k0, k1: 保留给操作系统的自陷/中断处理程序使用；用完后不必恢复原来的值；很少用于其它用途。

- gp: 用于两个不同目的。对于 Linux 程序使用的一类位置无关的代码 PIC，本模块外的代码和数据引用都要通过一个称为全局偏移量表 GOT(global offset table) 的指针表。gp 寄存器用来维护指向该表的指针。详情参阅第 16 章。

  在常规的非 PIC 代码（在简单的嵌入式系统中会用到）中，gp 有时用来指向一个链接时确定位置的静态数据中部。这意味着，利用 gp 作基指针，对于在 gp 指针前后 32K 范围内的数据存取，只需要一条指令就可完成。

  如果没有全局指针，存取一个静态数据区域的值需要两条指令：一条是获取由编译器和装载程序计算出的 32 位地址常量的高位，另外一条才真正存取数据。

要建立相对于 gp 的地址引用, 编译器在编译时刻必须知道一个数据链接后的内存地址在 64K 范围之内。实际上这是不可能知道的，只能靠猜测。通常的做法是把小的（八个字节以内的）全局数据项放在 gp 的区域内，如果还是太大，就让链接器发出警告。

- sp: 堆栈指针的上下移动需要显式的通过指令来实现，因此 MIPS 通常只在子程序入口和出口才调整堆栈指针；由被调用的子程序来负责完成。通常在子程序入口处把 sp 调整到子程序中间堆栈可能达到的最低点，从而编译器可以通过相对于 sp 的固定偏移量来存取堆栈变量。参见第 10.1 节关于堆栈的用法。

- fp: 也叫做 s8。如果由于某种原因编译器或程序员不能或者不愿计算相对于堆栈指针的偏移量，子程序可以用帧指针来记录/跟踪堆栈的情况。这种情况是有可能的，特别是当程序要涉及到运行时对堆栈作动态扩展的时候。有些语言明确支持这样做；汇编程序员总可以进行各种实验；用到库函数 alloca() 的 C 语言程序就是这样做的。

  如果堆栈的底部在编译时刻不能被决定，你就不能通过 sp 来存取堆栈变量，因此在函数开头，fp 初始化为相对于该函数栈帧一个常量的位置。巧妙地利用寄存器约定就能够使得这个行为只跟本函数有关，既不影响调用函数，也不影响任何嵌套的函数调用。

- ra: 在任何一个子程序入口处，ra 寄存器中保存着返回地址——因此典型的子程序都以一条 `jr ra` 指令结尾。理论上讲，这里可以使用任意寄存器，但是一些复杂的 CPU 因为采用优化技巧（分支预测），使得用 `jr ra` 时效果会更好。

  自身还要调用别的子程序的程序，必须首先保存 ra 的值，通常保存到堆栈上。

对浮点寄存器也有一个相应的标准，我们将在 7.5 节总结一下。我们在这里已经介绍了 MIPS 最初发布的寄存器的用法约定；近来新增了一些变化，老的约定就称为 o32 ——我们要直到 11.2.8 节才讨论更新的约定。

2.3 整数乘法部件及寄存器

MIPS 体系结构认为整数乘法非常重要，值得通过硬布线实现乘法指令。这一点在1980 年代的 RISC 芯片里并不普遍。另外一种做法用硬件实现能够匹配标准整数执行流水线的一个乘法步骤；对于复杂点的乘法，交给软件通过子程序(来模拟一个乘法操作)。早期的 Sparc CPU 就是这样做的。

还有一种用来避免设计整数乘法器复杂性的做法是在浮点单元中来运行乘法——Motorola 的昙花一现的 88000 CPU 家族就是提供了这样的解决方案——这样的缺点是违背了 MIPS 浮点协处理器作为可选的设计初衷。

与此不同，MIPS CPU 采用了一个专用的整数乘法部件，并没有集成到主流水线中去。乘法单元的基本操作是把两个寄存器大小的值相乘得到一个两倍大小的寄存器结果，存放在乘法单元里面。指令 **mfhi**、**mflo** 分两半将结果传送到指定通用寄存器。

因为乘法的结果返回没有快到能自动供随后的指令使用的程度，乘法的结果寄存器总是互锁的。在乘法运算完成之前任何企图读取结果的操作都将导致 CPU 停下来等待乘法操作结束。

整数乘法器也可以执行两个通用寄存器的除法操作。这时 **lo** 寄存器用来存放结果(商)，**hi** 寄存器用来存放余数。

你无法在一个时钟周期内得出乘法单元的结果：乘法要用 4–12 个周期，除法要用 20–80 个周期（取决于具体实现，对有些实现还取决于操作数的大小）。一些 CPU 拥有完整的或者部分流水线化的乘法单元——就是说，可以每一个或者两个周期启动一条乘除指令，尽管不能在四五个周期内得到结果。

MIPS32/64 包括了一条三操作数的 **mul** 指令，返回乘法结果的低半部分到一个通用寄存器中。但是那条指令必须停顿下来直到操作完成；高度优化调整的软件依然会用分离的指令分别计算乘法和提取结果。MIPS32/64 CPU（也包括目前市场上还有的其它 CPU）还有乘加操作，相继的乘法操作的积累加到 **lo/hi** 对中。

整数乘除操作从不产生异常：即使是除以 0 也不会发生异常（当然这样得到的结果是不确定的）。编译器常常会产生额外的指令检查和捕获错误，特别是被零除的错误。

还定义了指令 **mthi**、**mtlo** 用来将通用寄存器的值传送到乘法单元的内部寄存器。这对于从中断返回时恢复 **hi** 和 **lo** 的值是必不可少的，但在除此以外的其它地方可能都不会用到。

2.4 加载与存储：寻址方式

如前面所言，MIPS 只有一种寻址方式[1]。任何加载或存储操作的机器指令都可以写成：

```
lw    $1, offset($2)
```

你可以用任何寄存器作为目标和源寄存器。偏移量 offset 是一个有符号的 16 位的数字(因此可以取在 -32768 与 32767 之间的任何值)；加载所用的程序地址是 $2 寄存器的值与 offset 的和。这种寻址方式一般已足够存取 C 语言结构体的一个成员(偏移量是这个结构的起始地址到所要存取的结构成员之间的距离)。

[1] 已经不再完全正确了，现在有一种寄存器加寄存器寻址模式用于浮点数加载和存储

同时也支持实现以常量为索引的数组；用堆栈指针或者帧指针存取函数的局部变量；并为静态和外部变量提供一个以 **gp** 的值为中心的适当大小的全局空间。

汇编器提供一个类似简单直接寻址的方式来加载一个在连接时刻才能确定定地址的内存变量的值。

更复杂的寻址方式，如双寄存器寻址或索引变址方式，都必须通过多个指令的序列实现。

2.5 存储器与寄存器的数据类型

MIPS CPU 可以在一次操作中装载或存储一到八个字节。文档中采用的以及构成指令助记符的命名约定如下：

C 名字	MIPS 名字	大小(字节)	汇编助记符
long long	dword	8	ld 中的"d"
int	word	4	lw 中的"w"
long[2]			
short	halfword	2	lh 中"h"
char	byte	1	lb 中的"b"

2.5.1 整数数据类型

字节 byte 和半字 halfword 的加载有两种方式。符号扩展的指令 lb 和 lh 指令将数据值存放在 32 位寄存器的低位，并将高位用符号位(字节的位 7，半字的位 15)的值来填充。这样就正确地将一个有符号整数转换成了一个 32 位的有符号的整数，如下图中所示。

无符号的指令 **lbu** 和 **lhu** 用零来扩展数据，将数据值加载到 32 位寄存器的低位中，并用零来填充高位。

例如，假设寄存器 **t1** 中是存储器中一个 byte 字节宽度数据的地址，该处存放的值为 0xFE(有符号数 -2 或无符号数 254)，那么

```
lb      t2, 0(t1)
lbu     t3, 0(t1)
```

执行之后，寄存器 **t2** 的值为 0xFFFF FFFE(有符号数 -2)，**t3** 的值会是 0x0000 00FE（有符号数或无符号数 254）。

上面描述的是 32 位 CPU 的 MIPS 机器，但是许多机器都有 64 位的寄存器。结果表明是所有的部分字加载（甚至无符号数的加载）全都符号扩展到高 32 位；这个行为看上去古怪但却很有用，在 2.7.3 节中将详细解释。

短整数扩展到较长的整数的不同方式间的细微区别是由于 C 语言可移植性问题的历史原因造成的。现代的 C 语言标准定义了非常明确的规则来避免可能

的二义性。在象 MIPS 这样不能直接作 8 位和 16 位精度的算术运算的的机器中，包含 short 和 char 变量的表达式需要编译器插入额外的指令以确保数据该溢出时能够溢出；这种行为极为少用而且低效。当移植一个使用小整数变量的代码到 MIPS CPU 上的时候，你应该考虑找出哪些变量可以安全转换成 int 类型。

2.5.2 未对齐的加载和存储

MIPS 体系结构中，正常的加载和存储必须对齐；半字只能从双字节的边界加载；字只能从四字节的边界加载。一个对非对齐的地址的加载指令会导致自陷。因为 CISC 体系结构，象 MC680x0 和 Intel 的 x86 确实能够处理非对齐的加载和存储，当移植软件的时候，你可能会遇到这个问题；一个极端情况是你或许想安装一个异常处理程序来模拟相应的加载操作从而使它对用户程序保持透明——但是这样会使程序慢得难以忍受，除非加载非常少。

C 语言代码中声明的所有数据类型都会正确对齐。

当你事先知道你的程序需要从一个未知的、可能不对齐的地址传输数据时，MIPS 体系结构确实为此提供了一个双指令序列（要比通过一系列的字节加载、移位、相加快得多）。构成序列的指令其操作极为难懂，但通常都由宏指令 **ulw** (unaligned load word) 产生的。8.5.1 节将详细介绍。

还提供了一个宏指令 **ulh** (非对齐的加载半字)，由两个加载、一个移位和一个逐位“或”操作指令合成。

缺省情况下，C 编译器负责将所有的数据进行正确的对齐，但是在有些情况下（例如从文件中读取数据或与不同的 CPU 共享的数据）能够有效处理非对齐的整数数据是必须的。大多数编译器允许你标记一个数据类型可能是非对齐的，并会产生（比较高效的）特殊代码来处理——参见第 11.1.5 节。

2.5.3 内存中的浮点数据

从内存中将数据加载到浮点寄存器仅仅传送数据不做任何解释——可以加载一个无效的浮点数据(其实可以加载任意的位串组合)，直到对这些数据运算之前不会有任何浮点错误。

在 32 位处理器上，这允许你通过一个加载将一个单精度的数据放入一个偶数号的浮点寄存器中，你也可以通过一个宏指令加载一个双精度的数据，因此在一个 32 位的 CPU 上，汇编指令

```
l.d     $f2, 24(t1)
```

被展开为两个连续的寄存器加载：

```
lwc1    $f2, 24(t1)
lwc1    $f3, 28(t1)
```

在 64 位 CPU 上，**l.d** 是真正完成数据传输的机器指令 **ldc1** 首选推荐的别名。

任何一个遵循 MIPS/SGI 规则的 C 编译器都将八字节长的双精度浮点变量对齐到八字节的地址边界上。32 位硬件并没有这样的对齐要求，对齐仅仅是为了向后兼容：如果要 64 位 CPU 从一个未在八字节边界对齐的地址加载双精度数据，就会陷入异常。

2.6 汇编语言的合成指令

MIPS 的机器码书写起来可能相当枯燥太沉闷了；虽然从体系结构的原因我们有充分的理由说明为什么不能直接用一条指令来完成将一个 32 位的常量装入一个寄存器中，但是汇编语言程序员不想每次都得考虑这些琐碎的细节。因此 GNU 的汇编器(还有其它好的 MIPS 汇编器) 会为你合成一些指令。你只需要写一个 **li** (加载立即数)指令，汇编器会知道什么时候要生成两条机器指令。

这显然是很有用的，但也可能被滥用。有些 MIPS 汇编器对于体系结构的特点掩盖到了毫无必要的程度。在本书中，我们将尽量少用合成指令，当使用时会明确指出。另外，在指令列表中，我们一直会明确区分合成指令与机器指令。

我的感觉是合成指令是用来帮助程序员的，规范的编译器应该生成和机器代码严格的一对一的指令。[3]但是在这个不完美的世界里，还是有许多编译器实际上生成合成指令。

汇编器提供的帮助包括以下方面：

- 32 位立即数的加载：你可以写一个加载任何值(包括在链接时计算的内存地址值)的代码，汇编器将会把其拆开成为两个指令，分别加载这个数据的高半部分和低半部分。

- 从内存地址加载：你可以写一个加载驻留内存的变量的代码。汇编器通常会将这个替换成两条指令，先是一条把变量地址的高位加载至临时寄存器的指令，接着是一条以该变量的低位地址为偏移量的加载指令。当然这不适用于 C 语言函数内部定义的局部变量，局部变量是通过寄存器或堆栈实现的。

- 对内存变量的快速存取：有些 C 程序包含了许多对 `static` 和 `extern` 变量的引用，对它们加载/存储用两条指令开销太大了。一些编译系统通过有些运行时的支持避开了这一点。在编译/汇编时（最常见的做法是汇编器选择占用八个字节或更小空间的变量）选择一些变量，并将它们一起存放到最后大小在 64K 字节以内的一个内存区。然后运行时系统初始化一个寄存器——习惯上用 **$28** 也就是 **gp** ——来指向该内存区的中间位置。

  对这些变量数据的加载和存储现在就可以通过一条相对 **gp** 寄存器寻址的加载或存储指令来完成。

[3]在 2003/2004 年间对 GNU C 编译器的 MIPS 后端进行的重新改写，其背后奉行的就是这条原则。

- 更多类型的分支条件：汇编器合成了一整套根据两个寄存器的算术运算结果来进行条件分支的指令。
- 同一指令的简写或多种不同的写法：象 **not** 和 **neg** 这样的单目运算，是通过永远是零值的寄存器 **$0** 与操作数的 **nor** 和 **sub** 来实现的。你还可以把一个三个操作数的指令写成两个操作数的形式，汇编器将会把结果写到给出的第一个寄存器中。
- 隐藏分支延迟槽：在正常的情况下，汇编器如果看出将写在分支之前的指令移动到延迟槽没问题的话，就会这样做。因为汇编器能够看出的情况不多，所以并不太善于填充延迟槽。有一条汇编伪指令 **.set noreorder** 可以用来告诉汇编器不要自动移动指令，而由程序员人工控制。
- 隐藏加载延迟槽：有些汇编器会检测到紧接加载指令后试图使用加载结果的指令，如果行的话可能会上下移动一条指令。
- 未对齐的数据传送：非对齐的加载/存储指令（**ulh**，**ulw** 等等）能正确地存取半字和字数据，即使目标地址是未对齐的。
- 其它的流水线校正：一些指令（比如使用整数乘法单元的指令）在一些老的 CPU 上还有些额外的限制。如果你用的是这些老的 CPU，就会发现汇编器能帮上忙。

总之，如果你真的想要将汇编源代码（不在 **.set noreorder** 里头的代码）与内存中的指令对应起来，你需要借助反汇编工具。

2.7 MIPS I 到 MIPS64 ISA: 64 位(和其它)的扩展

MIPS 体系结构自从发明以来就一直在成长——最为显著的是从 32 位成长到 64 位。这个成长非常顺利，以致我们完全可以把当代的 MIPS 描述成 64 位的体系结构来，32位 只是其为低成本实现而明确定义的一个子集。本书没有这样做，是有几个原因。第一，实际发展的过程不是这样。如果这样描述，可能会有误导读者的嫌疑。第二，MIPS 给业界上的重要一课就是怎样平滑地扩展一个体系结构的艺术。第三，本书的材料其实是为 32 位 MIPS 而写的，后来经过扩充才包括了 64 位。

所以我们采用一个混合的方式。通常我们会先介绍 32 位的版本，一旦涉及到细节的时候，就会两个版本一起介绍。以后我们将用 ISA 代表指令集体系结构。

一旦 MIPS 开始演变，最初的 32 位 MIPS CPU (包括 R2000、R3000 及其其后继产品) 的 ISA 都相应的称为 MIPS I[4]。下一个广泛使用的 ISA 的变体做了

[4]这个好比一个影迷问你看没看过电影《终结者 I》，事实上从来没有一个电影叫那个名字。就连贝多芬的《第一交响曲》也曾被叫做《贝多芬交响曲》。

许多重要改进并为 R4000 及其后续产品上提供了完整的 64 位 ISA，称为 MIPS III。

尽管 ISA 有了许多演变，至少在应用层次上(当你在一个工作站上写应用程序时所有可见的代码)，新的指令集总能够向前兼容，完全支持为老的处理器写的程序。显然 32 位 ISA 不能运行 64 位程序；除此而外，唯一可能有问题的 ISA 版本就是 MIPS V，其中某些指令在 MIPS64 中没有。但是此后 MIPS V 也从来没有被实现。

MIPS II 曾经一度出现过，但是却无所作为，因为它的第一个实现 R6000 最后被 MIPS III R4000 后来居上而超过了。MIPS II 非常接近于 MIPS III 除去 64 位整数运算后的子集。对 MIPS II ISA 的这种理解在 1990 年代曾经有所回头，作为新实现的 32 位 MIPS CPU 的一个选择而出现。

如我们前面描述过的，不同的 ISA 演变定义了相应的内容；至少定义了在一个受保护的操作系统中用户程序可用的全部指令——包括浮点运算指令在内。[5] 与这些指令相应的，ISA 还定义和描述了整数、浮点数据和浮点控制寄存器。

在 MIPS32/64 之前，ISA 定义都非常小心的将 CPU 控制（协处理器 0）寄存器和整个 CPU 控制指令集排除在外。我不知道这样做有多大用处，当然这样由于信息不足确实可以为 MIPS 咨询业创造更多的工作机会；名叫《MIPS IV 指令集》的书对你要了解如何对 R5000 的高速缓存编程没有任何用处。

在实践中，协处理器 0 也伴随着正式的 ISA 一起演变。与正式的 ISA 类似，主要有两个版本：一个伴随着 R3000 ——现在基本上已经过时——另一个来源于最初的 MIPS III CPU ， R4000。R4000 族采用的方式现在已经作为 MIPS64 的一部分标准化了。我将称这两个分别称为“R3000 式“和“MIPS32/64“式的。但在本书的这一版不会有太多关于 R3000 式协处理器 0 的内容。

2.7.1 发展到 64 位

随着1990 年 MIPS R4000 的问世，MIPS 成为第一个投产的 64 位 RISC 体系结构的芯片。MIPS64 ISA[6] 定义了 64 位通用寄存器，一些 CPU 控制寄存器的宽度也不止 32 位。另外，所有的操作都产生 64 位的结果，当然有些从 32 位指令集继承过来的指令对 64 位数据的操作没有什么意义。在 32 位指令不能兼容的扩展到处理 64 位操作数的情况下，就增加新的指令。

尽管 MIPS32 CPU 允许采用只有 32 位浮点的 FPU，目前为止还没有这样做的。MIPS32 和 MIPS64 CPU 都采用真正 64 位浮点寄存器的 FPU，一次不再需要一对寄存器来容纳一个双精度的值。这个扩展与老的 MIPS I 模型（有 32 个 32 位寄存器成对使用这样看上去好像有 16 个 64 位寄存器）不兼容，所以在 CPU 控制寄存器中有一个模式开关可以设置成让寄存器的行为与 MIPS I 一样

[5] 但总是可以造出没有实现浮点的 CPU。

[6] R4000 和后来的 MIPS64 标准并不百分之百的兼容，但是很接近，我们本节把重点放在 MIPS32 和 MIPS64 的区别上。

以便允许用老软件。

2.7.2 谁需要 64 位?

到 1996 年，32 位已经不能提供足够大的地址空间给一些巨大的工作站和服务器应用程序。似乎权威人士一致认为程序的规模以指数的增长，每 18 个月左右就翻一番。按照这个增长速度，对地址空间的需求以每年$\frac{3}{4}$个位增长。真正的 32 位 CPU(68020, i386)出现并取代 16/20 位的机器是在 1984 年前后——确实 32 位在 2002 年左右真的显得不足了。如果这个数据让我们觉得 MIPS 1991 年的举动不够成熟，或许是对的——MIPS 最大的支持者 SGI 直到 1995 年才推广其第一个 64 位的操作系统。

当时有一种研究认为，为了使得一个对象可以在很长一段时间内通过其虚拟地址来命名，操作系统应当使用大范围的稀疏的虚拟地址空间。MIPS 早期的不成熟的举动就是受到了该项研究的促使。在被操作系统的发展速度所欺骗的机构中，比 MIPS 更知名的大有人在。Intel 统治世界的 32 位 CPU，在 Windows 95 操作系统将 32 位运算带入大众市场之前，就不得不等待了足足 *11* 年。

64 位体系结构一个连带的特点就是计算机一次可以处理更多的位，这可以加速一些图形和图像处理中数据密集型的应用程序。相对于以 Intel 的 MMX 为代表的多媒体指令扩充，这种做法是否更为可取尚未可知。后者通过一组新加的宽寄存器加宽了的数据通道，并能够在加宽的数据上同时处理多个单字节数据或多个 16 位数据。MIPS 的 DSP ASE 与此有些类似——见附录B.2节。

到了 1996 年，任何一个想要寻求长远发展的体系结构都需要有相应的 64 位实现。或许早点实现 64 位计算不算一个坏事。

MIPS 体系结构的与生俱来的天性——采用平坦的地址空间和通用寄存器用作指针——意味着 64 位寻址和 64 位寄存器是相伴的。即使在用不到 64 位宽地址的地方，寄存器宽度与 ALU 宽度的增加对处理大量数据的程序也非常有用，这在图形处理或高速通信应用当中是很常见的。

给 MIPS 体系结构(以及其它一些简单的 RISC 体系结构)带来的一个希望就是进入 64 位将使分段（即使在 64 位的版本中，分段仍然是 x86 和 PowerPC 体系结构一个负担）变得毫无必要。

2.7.3 关于 64 位与 CPU 模式切换：寄存器中的数据

将一个 CPU 扩展到新领域的标准做法就是，很久以前 DEC 公司从 PDP-11 升级到 VAX 、Intel 公司从 8086 升级到 i286 和 i386 的时候所用的办法：在新的处理器中定义一个模式切换位，当该位置位时，就让处理器就表现出同其前代产品一样的行为。

但是模式切换是一种凑合的做法，而且在没有微代码的机器中实现起来很难。MIPS64 走了另外一条路：

- 保留所有的 32 位体系结构的指令。
- 只要是仅仅运行 MIPS32 指令，就要保证 100% 的兼容：程序的行为完全相同，每个 MIPS64 的 64 位寄存器的低 32 位存放的值与相应的 MIPS32 寄存器在此时的值完全相同。
- 定义尽可能多的 MIPS32 指令，以便既能保持兼容性又能用做 64 位指令。

在这里，关键的决定(当你清楚这个问题后，就很简单)是，运行在 MIPS32 兼容状态下时，寄存器的高 32 位该存放什么值？选择倒是有不少，但简单而又真正有意义的就那么几个。

我们可以简单的决定寄存器高 32 位是未定义的；当运行在 MIPS32 兼容模式下时，寄存器的高 32 位可以含有任何旧的垃圾值。这个方法实现很简单但不能满足上述第三点：测试和条件分支指令(需要测试寄存器的值是否相等或者为负，得查看最高位——位 31 或位 63)需要各自不同的 MIPS32 和 MIPS64 版本。

第二种要好些的方案是，当运行 MIPS32 指令时，寄存器高位保持为零；但是这种方法同样要求对负数的测试和比较指令加倍。还有，MIPS64 的 `xor` 指令在两个高位为零的值上操作时，不能自然的产生一个高位为零的值。

第三种，也是最好的方案是将寄存器的高 32 位与第 31 位一样。如果(当仅运行 MIPS32 位指令时)我们保证每个寄存器存放着正确的低 32 位值并且高 32 位是第 31 位的复制，那么所有的 MIPS64 比较和测试指令都与其 MIPS32 的相应版本兼容——所以我们可以继续使用这些指令的 MIPS32 编码。所有的逐位逻辑操作指令也同样能用（对位 32 到 63，适用对位 31 同样的操作）。

这个正确/成功入选的方法可以这样描述为，将寄存器中 32 位的值符号扩展到 64 位；注意这种方法并没有考虑该 32 位的值是有符号的还是无符号的。

采用这个方案，MIPS64 需要新增简单算术指令的 64 位版本（MIPS32 的 addu 指令，当遇到 32 位溢出时，必须在低 32 位存放溢出的结果，并将第 31 位复制到高 32 位——这与 MIPS64 加法不一样!）。还需要新增 64 位的存储器加载和移位指令，但是增加得不多。当 64 位数据需要一个新的指令时，其指令助记符增加一个"d"表示"double"，生成象 **daddu**、**dsub**、**dmult** 和 **ld** 这样的名字。

略微不明显的是已有 32 位的加载指令 **lw**，现在确切的意思是加载一个带符号的字，因此引入一个新的高位零扩展的 **lwu** 指令。由于支持存取指令现有的各种变体的需要，以及(在常数移位指令情形) 支持采用不同操作码以摆脱 MIPS32 的 5 位的移位量限制的需要，新增的指令的数目还会进一步增加。

第 8 章详细列出了所有的 MIPS64 指令。

2.8 基本地址空间

MIPS 处理器对地址空间的使用和处理和传统的 CISC CPU 有着微妙的不同，我们知道这一点容易引起混乱。请仔细阅读本节的第一部分。我们将从最初的 32 位开始，然后再介绍 64 位——耐心点你会明白为什么这样做。

先看一些基本的原则。在 MIPS CPU 里，你写在程序中的地址绝不会和芯片的物理地址相同（有时变化很小，但并不相同）。我们分别称之为程序地址[7]和物理地址。

一个 MIPS CPU 可以运行在两种特权级之一上：用户模式和核心模式[8]。简单起见，我们常说用户态和核心态。但是 MIPS 体系结构的一个特点就是从核心态到用户态的变化并不改变操作的行为，只是有时某些操作被认为是非法的。在用户态，地址最高位为一的任何程序地址都是非法的并会导致自陷。还有，有些指令在用户态将会导致异常。

在 32 位下（图 2.1），程序地址空间划分为四大区域，每个区域有一个传统的（完全没有意义的）名字。根据地址所处的区域不同，处理也不同：

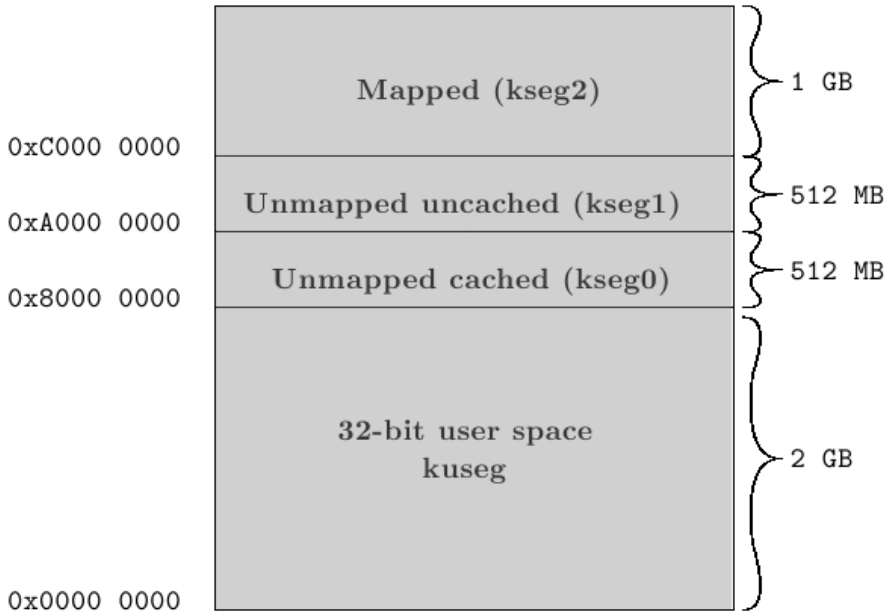

图 2.1: MIPS 存储器映射：32 位情形

[7]在这个意义下，程序地址和虚拟地址完全相同——但是对好多人来说虚拟地址这个词让人想到许多复杂的操作系统问题，与我们这里的讨论无关。

[8]在 R4000 以后的 MIPS CPU 有第三个管理模式；但是迄今为止所有的 MIPS OS 都忽略管理模式，我们多数时候也将忽略它。

kuseg 0x000 0000 - 0x7FFF FFFF (低端 2G)：这些地址是用户态可用的地址。在有 MMU 的机器里，这些地址将一概被转换（参见第 6 章）。除非 MMU 已经设置好，否则不应该使用这些地址。有些文档将这部分称为"useg"，特别是在描述用户程序看到的地址空间的时候。本书不再采用"useg"。

对於没有 MMU 的机器，这些地址的行为与具体实现机器相关。具体 CPU 的手册将会提供有关这方面的信息。如果想要你的代码能够移植到无 MMU 的处理器上，或者能够在不同的无 MMU 的处理器间移植，应避免使用这块区域。

kseg0 0x8000 0000 - 0x9FFF FFFF(512M): 只要把最高位清零这些地址就会转换成物理地址，映射到连续的低端 512M 的物理地址。因为这种转换极为简单，常常称这段地址为"非转换的"的区域，但其实不然。

该区域的地址几乎总是要通过高速缓存来存取，所以在高速缓存适当初始化之前，不能使用。这个区域在无 MMU 的系统中用来存放大多数程序和数据；在有 MMU 的系统中用来存放操作系统核心。

kseg1 0xA000 0000 - 0xBFFF FFFF(512M): 这些地址通过把最高三位清零的方法来映射到物理地址，重复映射到了低端 512M 的物理地址。但是这一次存取不经过高速缓存。

kseg1 是唯一的在系统重启时能正常工作的地址空间。这也是为什么复位时的入口点（0xBFC0 0000）放在这个区域。入口点相应的物理地址是 0x1FC0 0000 ——把这一点告诉你的硬件工程师[9]。

因此要使用这个区域去存取初始的程序 ROM；大多数人也把该区用作 I/O 寄存器。如果硬件设计人员提议要把这些东西映射到物理内存的低 512M 空间之外，你得劝阻他。

kseg2 0xC000 0000 - 0xFFFF FFFF (1G): 这块区域只能在核心态下使用并且要经过 MMU 的转换。在 MMU 设置好之前，不要存取该区域。除非你在写一个真正的操作系统，否则来说没有理由用 kseg2。

有时你可能会看到该区域被分成两半，分别叫做 kseg2 和 kseg3，意在强调其低半部分 (kseg2) 可供运行在管理态的程序使用。要是你真有可能用到管理模式的话...

2.8.1 简单系统的寻址

MIPS 的程序地址从来不会和物理地址简单相等。但对於简单的嵌入式软件而言可能只用 kseg0 和 kesg1 的地址，它们和物理地址有着非常简单的映射关系。

[9] 你的硬件工程师不会是第一个把 ROM 置于物理地址 0xBFC0 0000 然后发现系统无法启动的人。

从 0x2000 0000（512M）开始向上的物理地址空间在上述简单情形下没有任何的映射，大多数简单的系统所有地址都映射到 512M 以下。但是如果真的需要，你可以通过设置存储器管理单元的（TLB）的方式，或者使用 64 位 CPU 的一些额外空间，来存取 512M 以上的物理地址。

2.8.2 核心与用户特权级

在核心特权级下（CPU 启动时），可以作任何事情。在用户态下，2G（最高位置位的）以上的程序地址是非法的，会导致陷入异常。注意的是，如果 CPU 有 MMU，这意味着所有的用户地址在真正到达物理地址之前必须经过 MMU 的转换，从而使得操作系统有能力可以防止用户程序失去控制/乱冲乱撞。同时也意味着对一个运行着没有地址转换的 OS 的 MIPS CPU 来说，用户特权级其实是多余的。

另外，在用户态下，有些指令——特别是操作系统需要的那些 CPU 控制指令——是非法的。

要注意的是，当你改变核心态/用户态特权级模式位时，并不改变对任何行为的解释——仅仅意味着某些功能在用户态下不允许了。在核心态下，CPU 可以存取低位地址，就和在用户态一样，并且地址也是做同样的转换。

还要注意的是，尽管听上去好象是在说“核心态是操作系统用的，用户态是日常简单的代码用的”，但是事实与此相反。有些简单的系统（包括许多实时操作系统）全部代码都是运行在核心态下的。

2.8.3 完整的图像：64 位的地址映射

MIPS 地址总是通过一个寄存器的值加上一个 16 位的偏移量形成的。在 64 位 MIPS CPU 里，寄存器中总是 64 位的值，因此就有 64 位的程序地址。这样巨大的地址空间允许我们在划分地址空间是可以无所顾忌，从图 2.2 中可以看到地址空间是怎样划分的。

首先要注意的是 64 位存储器映象是包含在 32 位映象里面的。这是个奇怪的技巧——就象《Dr. Who》中的 TARDIS，里面要比外面大得多——而且依赖于我们在 2.7.3 节的规则来实现：当模拟 32 位指令集的时候，寄存器存放的是 32 位值的 64 位符合扩展。这样结果就是，32 位程序可以存取 64 位程序空间的最低和最高的 2 GB。所以扩展的映像把地址空间的最低和最高区域分配给和 32 位下一样的用途，扩展的地址空间位于这两者之间。

大大扩展的用户态和管理态的地址空间在实际应用中不可能有太大的用处，除非你在实现一个虚拟存储的操作系统；因此许多 MIPS64 的用户仍然把指针定义为 32 位的目标地址。那些的大块的无需地址转换的物理内存窗口可以用于克服 kseg0 和 kseg1 的 512M 的限制，但是也可以通过对存储器管理单元（TLB）编程来达到同样效果。

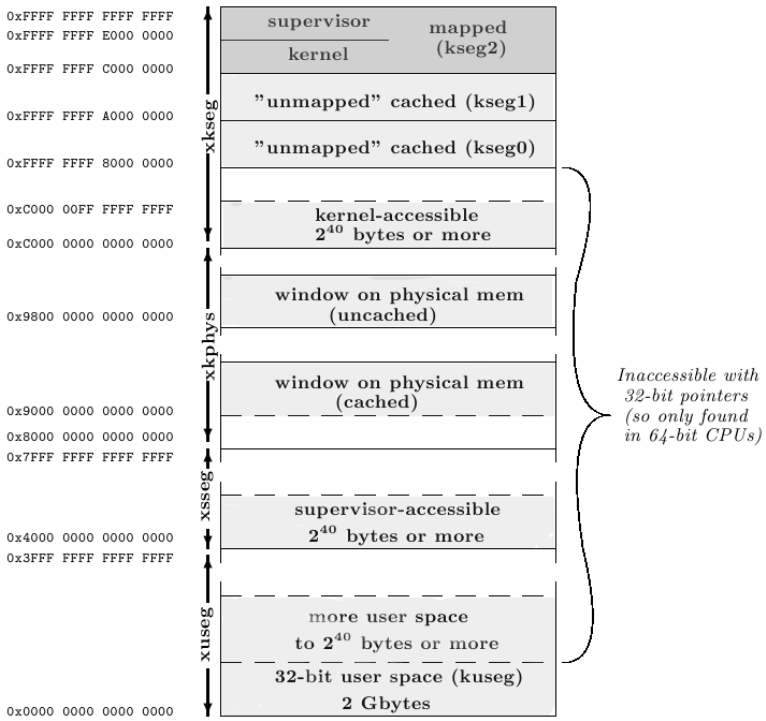

图 2.2: 64 位的存储器映射

2.9 流水线可见性

任何一个有流水线的 CPU 硬件对於那些不能满足严格的一个时钟周期规则的操作都将会存在时间延迟。体系结构的设计者要选择这些其中的哪一些（如果有的话）延迟对程序员是可见的。隐藏时序特性简化了程序员对 CPU 的编程模型，但这将复杂性加到了硬件实现人员身上。将时序调度问题留给程序员和软件工具简化了硬件实现，但同时又会给软件开发和移植带来问题。

我们已经提过几次，MIPS 体系结构让一些流水线的特点是可见的，把任务交给程序员或者编译器负责。下面几点总结了关于流水线效果可见的情形：

- 分支延迟：在所有的 MIPS CPU 里，紧跟着分支指令后的指令（即分支延迟槽中的指令）即使跳转成功也会执行。程序员或者编译器应当找出一条有用的或者至少是无害的指令填充延迟槽——最差的情形是填上 `nop` 。但是，除非明确指定，就连汇编器也会隐藏分支延迟的。

  在 MIPS II 指令集中作为一个可选项引入的“可能分支*(branch-likely)*”指令中，延迟槽中的指令仅在分支被接受的情况下被执行；用到该指令的情形可参见 8.5.4 节。

- 加载延迟：我想不起来有任何的 MIPS CPU 其中紧跟在加载之后的指令能够没有延迟地使用刚刚加载的数据。[10]

  但是优化的编译器和程序员总应该知道一个具体的 CPU 需要多少时间把数据准备好能用——即加载至使用的延迟。太长的加在使用延迟影响性能，硬件设计者会做许多工作以确保数据能在装载只后的下下一条指令可用。

- 浮点数(协处理器 1)的问题：浮点运算几乎都要花费多个时钟周期来完成，典型的 MIPS FPU 硬件通常有多个相对独立的流水线部件。MIPS 硬件必须把 FPU 流水线隐藏起来；允许浮点计算与其后续的指令并行的执行；当一个指令读取一个尚未完成的浮点计算的结果寄存器时，CPU 就会阻塞。真正重量级的优化需要编译器有指令重复比率表和各种目标 CPU 的延迟表等。但是，你根本不会想要你的程序运行依赖这些。

- CPU 控制指令问题：这个部分是需要慎重对待的地方。当你改变 CP0 的域比如 CPU 状态寄存器的内容时，你潜在地影响到发生在流水线所有阶段的东西。

  有了 MIPS32/64（至少第二版）之后，情况好多了。与 CP0 的交互被分成了两部分。那些前一个 CP0 操作可能会影响后一个指令的取指的情形是最为麻烦的，称之为指令遇险*(instruction hazard)*；其余情形称为执行遇险*(execution hazard)*。随之就提供了两种不同风格的遇险防护*(hazard barrier)*指令：一条适合于执行遇险的防护指令，以及能让你免于指令遇险的若干条增强的分支指令。如果在生产者和消费者之间放上合适的遇险防护指令，就能能够保证你不受部分完成的 CP0 的副作用影响。参阅第 3.4 节。

  在 MIPS32/64 第二版之前，与 CP0 的打交道的任务完全交给了机器相关的特性。没有正式提供可用的遇险防护指令，程序员必须阅读 CPU 手册以找出怎样加上足够多的填充指令以确保 CP0 控制的副作用有足够时间传播。

[10] 在很老的 MIPS I CPU 里，要由程序员或者编译工具负责安排一条空操作 no-op，但是你也许永远不需要再担心这种情况了。

第 3 章　协处理器 0：MIPS 处理器控制

除了通常的运算功能之外，任何处理器都需要一些部件来处理中断、配置选项以及需要某种机制来监控诸如片上高速缓存(cache)和定时器等功能。ISA 对运算指令集的处理方式很漂亮，做到了与实现无关；但是对处理器控制还想要做到那么漂亮可就难了。

如果我们把不同的功能分到几章里分别介绍，你们可能更乐于也更易于接受。我们下面就是这样做的。但是在介绍之前我们有必要先了解一下实现这些特性的公共机制。在触及本书后续的三章内容之前，建议读者先要仔细阅读本章的第一部分；特别要注意下一页介绍的协处理器一词的用法。

说了这么多，那么到底 MIPS CPU 上的协处理器 CP0 干些什么工作呢？

- CPU 配置: MIPS 硬件常常是很灵活的，您可以选择 CPU 的主要特性（例如第 10 章节所述的尾端），或者改变系统接口的工作方式。这些选项的控制和可见性由一个或多个内部寄存器提供。
- 高速缓存控制: MIPS CPU 总是集成了高速缓存控制器，除了最古老的芯片外也都集成了高速缓存本身。CP0 的 `cache` 指令用来——以多种不同方式——操纵高速缓存的数据块。我们将在第 4 章讨论高速缓存。
- 异常/中断控制: 异常或者中断发生时的行为以及怎样处理，都是由 CP0 控制寄存器和几条特殊指令来定义和控制的。这在第 5 章讨论。
- 存储管理单元控制: 在第 6 章讨论。
- 杂项: 总有更多的东西：定时器、事件计数器、奇偶校验、错误检测等等。每当新增的功能被集成到 CPU 里边，不能再方便地当作外设访问时，这里就成为它们的归属。

MIPS 对协处理器一词的特殊用法

*协处理器*一词通常用来表示处理器的一个可选部件，负责处理指令集的某个扩展。MIPS 标准指令集略去了实际的 CPU 需要的很多功能，但是为多达四个的协处理器预留了操作码和相应的指令域。

其中之一（协处理器 1）是浮点协处理器，这就是真正公认的意义上的协处理器。

另一个（协处理器 0 或者说 CP0）是 MIPS 所谓的*系统控制协处理器*，其指令对处理用户态程序职责之外的所有功能都是必不可少的；这也是本章讨论的主题。

协处理器 0 是不能独立存在的，这并不是说它是可有可无的——比方说，你不可能造出一个没有状态寄存器的 MIPS CPU。它的确提供了一种对访问状态寄存器的指令进行编码的标准方法。这样即使状态寄存器的定义在不同的 MIPS 实现之间发生了变化，至少可以保证同样的汇编程序还能在变化前后的两种 CPU 上用。

仅用于 OS 的协处理 0 的功能被有意地从 MIPS ISA 圈离开来。在 MIPS 的早期，系统供应商毫无例外地都提供一个定制的操作系统内核，这样 CP0 的变化需要对内核做相应地修改——但是仅限于内核；并不影响应用程序的兼容性。所以从 MIPS I 到 MIPS V， CP0 的功能都被认为是依赖于具体实现的。

情况已经不再如此了。现在有多个团体在为 MIPS 构建操作系统代码，要是操作系统在不同的 MIPS CPU 之间还必须要移植才行的话，那可真够让人头疼的。所以新的标准 MIPS32 和 MIPS64 充分详细地定义了 CP0 的寄存器和功能，以便足够构建一个可移植的操作系统。

还是回到体系结构上来：在四个协处理器编码空间中，CP3 已经被 MIPS32/64 浮点指令侵占，现只在确信绝对不会实现浮点的时候才能使用。CP2 还是可用的，偶尔用于定制的 ISA 扩展或者在几个 SoC 应用中提供专用的寄存器。CP1 就是浮点单元自身。

我们会在本章后半部分总结所有在 MIPS32/64 CPU 中能找到的东西。但是暂时让我们先把这些功能放到一边，看看用于实现的机制吧。MIPS CP0 指令为数并不多——只要可能，对 CPU 的底层控制都是对特殊的 CP0 寄存器位域的读写实现的。

表 3.1 介绍了 CPU 控制寄存器的功能。表中列出了符合 MIPS32/64 标准必定要实现的每个寄存器，此外还有几个常见的可选寄存器。

这不是一个完整的列表；其它寄存器与可选的指令集扩展（ASE）有关，或者与 MIPS32/64 别的可选特性有关。

此外，MIPS CPU 可能还会有一些与具体实现相关的寄存器——这是给体系结构增加新特性的推荐做法。请参考具体的 CPU 手册。

为了避免在这个时候就让你陷入一堆的细节，我们单列了一节：3.3 节，专门讲述按照 MIPS32/64 规定必须具备的 CP0 寄存器的每一位的细节。如果仅对该节后面的章节有兴趣，现在可以暂时跳过那一节。

我们列这些寄存器的时候，**k0** 和 **k1**（通用寄存器 **$26–27**）值得一提。这是两个（由软件约定）预留下来的用于异常处理代码中的通用寄存器。预留至少一个通用寄存器是必要的；而具体预留哪一个则是随意的，但预留的寄存器必须是被所有现存的 MIPS 工具包和二进制程序中都接受的。

表 3.1: MIPS CPU 控制寄存器

寄存器助记符	CP0寄存器编号	描述
SR	12	状态寄存器(Status Register)，一反常规的基本上由可写的控制位域组成。包括确定 CPU 特权等级、哪些中断引脚使能和其它的CPU 模式等位域。
Cause	13	什么原因导致异常或者中断？
EPC	14	异常程序计数器（Exception Program Counter）：异常/中断结束后从哪里重新开始执行。
Count Compare	9 11	这两个寄存器一起形成了一个简单而有用的高分辨率定时器，频率（通常）为 CPU 流水线频率的一半。
BadVaddr	8	导致最近的地址相关异常的程序地址。即使没有 MMU，各种地址错误也都会设置它。
Conext EntryHi EntryLo0–1 Index PageMask Random Wired	4 10 2–3 0 5 1 6	对存储器管理和地址转换硬件(TLB)编程的寄存器，这在第 6 章讲到。
PRId	15	CPU 类型和版本号。类型号由 MIPS 公司管理，（至少）当 CP0 寄存器集变动时应当不同。表 3.3 中列出了到 2004 年中为止用过的值。
Config Config1-3	16 16.1–3	CPU 参数设置，通常由系统决定；一些可写，另一些只读。有些 CPU 有高编号的专用寄存器。
Ebase	15.1	异常入口点基址和——多 CPU 系统的——CPU ID。

表 3.1 续

寄存器助记符	CP0寄存器编号	描述
IntCtl	12.1	设置中断向量和中断优先级
SRSCtl SRSMap	12.2 12.3	影子寄存器控制，参见第 5.8.6 节。 当 CPU 使用“向量化中断”特性的时候，八个影子寄存器编号到八个可能的中断原因之间的映射。
CacheERR ECC ErrorEPC	27 26 30	分析(甚至可能恢复)内存错误的域，适用于在数据通路上使用错误校验码的 CPU。详细信息参见第 4.9.3 节。
TagLo DataLo TagHi DataHi	28.0 28.1 29.0 29.1	用于高速缓存控制的寄存器，第 4.9 节会讲到。
Debug DEPC DESAVE	23.0 24.0 31.0	用于 EJTAG 调试单元的寄存器，在第 12.1.7 节讲到。
WatchLo WatchHi	18.0 19.0	数据观察点工具，当 CPU 试图在该地址存取数据时会发生异常——对调试有潜在的用途，参见第 12.2 节。
PerfCtl PerfCnt	25.0 25.1	性能计数寄存器（后续的奇偶号可以选择更多的控制/计数寄存器对）。参见 12.4 节。
LLAddr	17.0	有些 CPU (主要是那些带有一致性缓存或者多线程扩展的 CPU）存放一个与 ll(load-linked) 指令相关的地址；地址存入后是可见的，虽然只有诊断软件才会去读它。参见第 8.5.2 节。
HWREna	7.0	一个可写的位图，决定哪些硬件寄存器可以被用户态程序访问——参见第 8.5.12 节。

3.1 CPU 控制指令

有几条特殊的 CPU 控制指令用于实现存储管理，我们把这些留到第 6 章。MIPS32/64 定义了一组 **cache** 指令来实现管理高速缓存必须的所有操作，这在第 4 章会中介绍。

除此之外，MIPS CPU 控制只需要很少几条指令。先从统一访问我们刚才列出的那些寄存器的指令开始：

```
mtc0    s, <n>      # 把数据传送到协处理器 0
```

这条指令把 CPU 通用寄存器 s 的内容传送到协处理器 0 寄存器 n，数据为 32 位（即使在 64 位CPU 里，很多 CP0 寄存器也只有 32 位长，但对于少数长的 CP0 寄存器有一条 **dmtc0** 指令）。这是设置 CPU 控制寄存器的唯一方法。

当 MIPS 刚刚出现的时候，最多可以有 32 个 CP0 寄存器。但是 MIPS32/64 可以允许多达 256 个寄存器。为了保持指令向前兼容，这是通过在 CP0 号（实际上是指令中以前编码为零的域）后附加 3 位的 *select* 域来实现的。这样 **mtc0 s, $12, 1** 就解释为存取“register 12，select 1。”我们写做 12.1。

在汇编程序里直接引用控制寄存器的编号来是不良习惯；通常应该使用表 3.1 中的助记符。大多数工具链把这些名字定义在一个 C 风格的 `include` 文件里，然后用 C 的预处理器作为汇编器的前端；参考工具包的文档找出具体做法。虽然原始的 MIPS 标准有相当的影响力，但是对这些寄存器的命名还是各有不同。本书将一直使用表 3.1 中的助记符。

从 CP0 控制寄存器中取出数据与之相反：

```
mfc0    d, <n>    # 从协处理器 0 取出数据
```

通用寄存器 d 装入 CPU 控制寄存器 n 的值。这是查看控制寄存器值的唯一方法（类似也有一个 dmfc0 变体用于少数 64 位宽的寄存器）。这样，如果您想要更新控制寄存器内部的单个域——比如说——状态寄存器 SR 吧，通常代码是这个样子：

```
mfc0    t0, SR
and     t0, <要清零的位的反码>
or      t0, <要置 1 的位>
mtc     SR, t0
```

控制指令集最后一个重要部分是一种消除异常效果的方法。我们会在第 5 章详细讨论异常，但基本的问题是每个实现任何一种安全操作系统的 CPU 都要共同面对的；这个问题就是异常可以在运行用户(低特权级)代码时发生，但是异常处理运行在高特权级。[1] 因此当从异常返回用户程序时，CPU 需要避开两种风险：一方面，如果在控制返回用户程序之前特权级就降低了，马上就会因为违反特权级而得到一个致命的二次异常；另一方面，如果在降低特权级之前先回到用户代码，那么一个恶意的无特权的程序就有可能得到机会以内核特权级运行一条指令。从编程的角度看来，返回到用户态和改变特权级的操作必须是不可分的（用计算机的行话来说，就是*原子的*）。

在除了最古老的之外，所有的 MIPS CPU 都是用指令 **eret** 来完成这个任务（在那些失传已久的原始 CPU 上，需要一条转移指令后接一条 **rfe** 作为延迟槽）。这个问题我们在第 5 章将详细讨论。

[1] 用软件触发的异常——系统调用——作为用户代码请求（运行在高特权级上的）操作系统内核服务的唯一机制，几乎是所有 CPU 的通用做法。

3.2 什么时候要用到哪些寄存器？

在下面这些情况你需要和这些寄存器打交道：

- 上电后：需要设置 **SR** 来使 CPU 进入一个可工作的状态，以便能够顺利执行随后的引导过程。硬件通常的做法都是在复位后让许多寄存器的位为未定义。

  除了最早期的之外，MIPS CPU 都有一个或者多个 **Config** 寄存器：**Config** 和 **Config1-3**。有些 CPU 可能更多，有一些专用的域（**Config7** 有时用作给具体 CPU 专用的域）。

  第一个 **Config** 寄存器有几个可写的域可能需要先设置，否则做不了多少事情。请和您的硬件工程师商量，确认 CPU 和系统对于配置寄存器的设置达成一致，至少可以启动到能写这些寄存器这一步！

- 处理任意异常：在早期的 MIPS CPU 中任何异常（除了一个特殊的 MMU 事件之外）都调用一个固定入口地址的公共的“通用异常处理程序”。但是自那以后的很多年间，有越来越多的理由支持对不同的目的使用分开的异常处理程序；参见第 5.3 节。

  在异常入口处，不保存任何程序寄存器，只有返回地址被存在 **EPC**。MIPS 硬件对于堆栈一无所知。在任何情况下一个安全操作系统的拥有特权的异常处理程序不能假定用户级代码的正确性——特别地，它不能假定栈指针有效或者栈空间可用。

  你至少需要用 **k0** 和 **k1** 中一个来指向为异常处理程序预留的一些内存空间。然后就可以保存东西，必要时还可以用另一个来从控制寄存器倒腾数据。

  通过察看 **Cause** 寄存器可以找出发生的是哪种类型的异常并做相应处理。

- 从异常返回：控制最终必须返回到在异常入口处保存到 **EPC** 中的地址。不管是什么异常，返回时都要把 **SR** 寄存器调整回原来的值、恢复用户态特权、允许中断以及消除异常的一般影响。最后，异常返回指令 **eret** 合并完成了返回用户空间和复位 **SR(EXL)** 的功能。

- 中断：**SR** 用来调整中断掩码，决定哪些（如果有的话）中断被赋予比当前更高的优先级。硬件没有提供中断优先级，但是软件可以随意。

- 纯粹为了引发异常的指令：这些指令常用于（系统调用、断点调试以及某些指令仿真等）。所有的 MIPS CPU 都实现了 **break** 和 **syscall** 指令；有些实现还加了额外的指令。

3.3 CPU 控制寄存器及其编码

控制寄存器的编码

> 说到这里有必要对保留域说明一下。许多未用的控制寄存器域被标记为“0”。这些域里的位保证读出为零，写入也没有什么害处（当然写入的值会被忽略）。另一些保留域被标记为“reserved”或者“x”；这时就要小心了，这些域永远应当写入零或者写入前面读出的值，但不应当认为读出的值一定是零或者其它具体值。

这一节讲述控制寄存器的格式并概要描述各个功能域。大多数情况下，可以在后面几节找到更详细的内容。我们把有关存储管理的寄存器留到第 6 章，把仅用于高速缓存管理的寄存器留到第 4 章，那里我们将全面地讲解高速缓存。

注意具体的 CPU 可能会在某些寄存器中定义一些额外的域。这里叙述的要么是 MIP32/64 强制规定都必须有的，要么是看上去很通用的。

3.3.1 状态寄存器（SR）

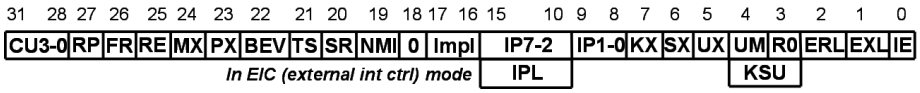

图 3.1: SR（状态）寄存器的各个域

MIPS CPU 有极少的几个模式位；仅有的几个定义在高度紧凑的状态寄存器 SR 的域中，如图 3.1 所示。这是 MIPS32/64 标准定义的全部位；有些空余的域可用于具体实现相关的用途。

我们再次强调，MIPS CPU 里没有不进行地址转换(nontranslated)或者不进行高速缓存（ noncached ）的模式；是否进行地址转换或者经过高速缓存是程序地址决定的。

下面是关键的公有域；新实现中把任何一个用于任何别的用途都是很不好的，从现在到可以预见的将来这些域的用法很可能都不会变化。

CU3–0 每个分别代表协处理器 3–0 的“协处理器使能”。协处理器 1 是浮点处理单元 FPU ——设置为 1 代表有浮点处理器并且使用，设置为 0 则代表禁止 FPU。当设为 0 的时候，所有 FPU 指令都会导致异常。没有浮点硬件时把它设为 1 显然不行；但有浮点硬件时（设为 0）关掉浮点部件可能会有用。[2]

[2] 为什么好好的浮点部件要关掉呢？有些操作系统对所有的新任务禁止浮点指令；当该任务试图使用浮点时，操作系统会捕获到异常并为它使能浮点部件。这样，我们可以区分出那些从不使用浮点的任务。在任务切换时，我们不需要为那些任务保存和恢复浮点寄存器，这样可以在关键的上下

RP 减小功耗（reduced power）——具体怎样做取决于 CPU。比如，可以降低 CPU 的运行频率、电压或者同时降低二者。具体情况请阅读 CPU 手册和咨询系统设计人员。

FR 模式切换：设为 1 则全部 32 个双倍宽度的浮点寄存器软件都可见；设为 0 让它们模拟 MIPS I 的成对 32 位浮点寄存器的行为。

RE (反转用户态下的尾端设置)：MIPS 处理器可以在复位时配置为任何一种尾端(如果不明白什么意思请参见 10.2 节)。由于人们总是很固执，现在 MIPS 实现分成了两个世界：DEC 和 Windows NT 是小尾端的；SGI 和它们的 UNIX 世界是大尾端的。嵌入式应用最初显得强烈倾向于大尾端，但现在已经彻底混淆了。

能运行来自另一个“世界”的软件对于操作系统会是一个有用的特性；RE 位使得这成为可能。当 RE 有效时，用户态的软件运行起来就好象 CPU 是配置为相反的尾端一样。然而，真正要达到跨“世界”运行需要软件上做很大的努力，到目前为止还没有人去做过。

MX 使能 DSP 或者 MDMX ASE（指令集）——同一个 CPU 不可能两者都有。在本书写作时，DSP ASE 还相当的新，而 MDMX 似乎缺乏工具链和中间件支持。

PX 参见下面对 SR(UX) 的描述。

BEV 启动时异常向量：当 BEV==1 时，CPU 采用 ROM(kseg1) 空间的异常入口点(参见 5.3 节)。运行的操作系统里，BEV 通常设置为 0 。

TS TLB 关闭：详细的信息参见第 6 章。在某些 CPU 上，如果一个程序地址同时匹配两个 TLB 表项（这是操作系统软件出了某种严重错误的标志），TS 位就会被置 1。在一些实现中，在这种状态下操作过久有可能导致内部竞争而损坏芯片，所以 TLB 就会关闭而停止匹配任何地址。TLB 关闭是一个不可逆的过程，一旦关闭，只有硬件复位才能清除。

一些 MIPS CPU 的 TLB 硬件十分简单因而不会出现这种情况，这样就不用实现这一位，另一种做法也可以只是表明试图写入重复表项的行为被阻止了。请参考你的 CPU 手册。

SR、NMI 软复位或者不可屏蔽中断发生了：MIPS CPU 提供几种不同方式的复位，以硬件信号相区别。特别是，配置寄存器 **Config** 在软复位过程中保持不变，但是在硬复位后必须重新编程。MIPS 的复位有点类似异常——当然软复位或者硬复位是一种 CPU 永远不会返回的异常——复位复用了许多异常处理机制。

文切换代码中节省点时间。

SR(SR) 域在硬件复位（所有的运行参数都要重新装入）后清零，而在软复位或 NMI 后置位。SR(NMI) 位只在 NMI 异常之后才置位。

IM7–0 中断屏蔽：一个 8 位的域定义允许哪些中断源活动时产生异常。其中六个中断源由 CPU 核外部的信号产生（一个可能由 FPU 使用，FPU 尽管位于同一个芯片上，但是逻辑上当成是外部的）；其余两个是 **Cause** 寄存器中软件可写的中断位。

如果你用的是新式 CPU 的 EIC 中断设置，那么对 **SR(IM)** 的解释就会与此不同；参见第 5.8.5 节。

除非你正在使用的是 EIC 系统，否则不提供中断优先级：硬件对待所有中断位都一样。详情参阅第 5.8 节。

UX,SX,KX 广义的讲这些使能 64 位 CPU 上的大得多的地址空间，三个不同的（用户、管理、核心）特权级各自有不同的位。当相应位置位后，最为常见的内存地址转换异常（TLB 未命中）被重定向到不同的入口点，在那里软件将处理 64 位地址。

此外，当 **SR(UX)** 为 0 时 CPU 将不在用户态下运行 MIPS64 ISA 中的 64 位指令。这使得操作系统可以构造一个用户态的“沙盒”，在其中 32 位程序——即使是一个有缺陷的程序执行对 32 位 CPU 来说非法的指令——其行为就和在 MIPS32 CPU 上执行完全一样。光靠这些特性的组合并非总能满足要求：比如你想在用户态用 64 位指令但是仍然坚持 32 位寻址，这时你就可能要设置 **SR(PX)**。

KSU CPU 特权级：0 是核心级，1 是管理级，2 是用户级。如果紧跟异常之后 EXL 或者 ERL 被置位，不管这个域是什么值 CPU 就都自动处于核心态。管理特权级是 R4x00 引入的，但从来没有用过。边栏的小字部分提供了一些背景资料。

有些手册的文档把该域记载为两个单独的位，高位叫做 **UM**。

ERL 错误级：在 CPU 发现收到错误的数据时该位置位。MIPS CPU 可以选择对从高速缓存或者内存收到的数据块进行额外的奇偶校验位或者 ECC（纠错码）位检查。奇偶校验错一般都是致命错（除非有已知良好的数据可供替换）。但是 ECC 错误只要错误位不超过一位（或两位）则可以通过软件纠正。

当这种错误出现时，CPU 收到一个奇偶校验/ECC 错误异常，并且该位置位。对这个异常的处理和标准异常是分开的，因为可纠正的 ECC 错误可能发生在任何地方——甚至发生在常规异常例程最为敏感的部分——如果系统试图纠正 ECC 错误接着运行，就必须在任何地方都能修复错误。做到这点太有挑战性了，因为异常处理程序连一个可以安全使用的寄存器都没有；而没有个寄存器用做指针，就没法开始保存寄存器值。

为了摆脱这个困境，对 **SR(ERL)** 赋予了一种特殊的任务，一旦其置位，就会深度影响处理器的行为；对正常的用户空间转换的地址的所有访问都将消失，从 0 到 `0x7FFF.FFFF` 的程序地址变成了一个映射到相同物理地址的不做高速缓存的窗口。目的是让高速缓存错误异常处理程序可以用 **zero** 寄存器来做基址，用基址+偏移量的方式来访问某些（由操作系统为此预留的）不经过高速缓存的内存空间，以便保存寄存器，并给自己腾出空间来运行。

EXL 异常级：任何异常发生时置位，这会强行进入核心态并禁止中断；目的是把 EXL 位维持足够长的时间以便软件决定新的 CPU 特权级和中断屏蔽该设成什么。

IE 全局的中断使能位：请注意不管这位是什么值，EXL 或 ERL 总是禁止所有的中断。

为什么有个管理态呢？

R3000 CPU 只提供两个特权级，这已经能满足绝大部分 UNIX 实现的要求，也是任何 MIPS 操作系统真正用到过的。那么为什么 R4000 的设计者要费很大的劲去加上一个从来没有用过的特性呢？

在 1989-1990 年的时候，MIPS 最大的成功之一就是在 DEC 公司的 DECstation 产品线上使用了 R3000 CPU，MIPS 公司想让 R4000 被选为 DEC 将来的工作站的 CPU。竞争者是 DEC 公司内部开发的后来发展成 Alpha 体系结构的 CPU，但那是从后面赶上来的；R4000 面世比 Alpha 大约要早 18 个月。

不管选择什么 CPU，它必须不仅能够运行UNIX，而且要能运行 DEC 的小型机操作系统 VMS。

Alpha 的基本指令集和 MIPS 几乎完全相同；最大的不同是试图取消部分字的存取操作（后来证明这是 DEC 的失误，又不得不改回来了）。

最终，看起来是 VMS 软件团队决定了选择 Alpha 而不是 R4000 ——他们坚持认为 R4000 过于简单的 CPU 控制系统会使得 VMX 太不安全，或者要花费很长时间移植。在这中间，他们特别举出了 MIPS 的两级安全性是个问题，R4000 的管理态就是 MIPS 公司对此的回应。

我很怀疑这个理由；我觉得这更多的是由于根深蒂固的偏见。技术问题只是一个烟幕弹，其背后掩盖的是 DEC 想要自己控制处理器开发的意图。DEC 认为自己控制处理器的开发至关重要，也许这对 DEC 来说没错。但是设想一下，要是 DEC 当初采用了 R4000 并且利用了这 18 个月的领先时间，事情的结局又该会是另外一番怎样的景象呢？

3.3.2 原因寄存器(Cause)

图 3.2 显示了 **Cause** 寄存器的各个域，用它来找出发生的异常类型，决定调用哪个异常处理例程。**Cause** 寄存器是异常处理的关键寄存器，从最早期的 MIPS CPU 以来几乎从未变化。

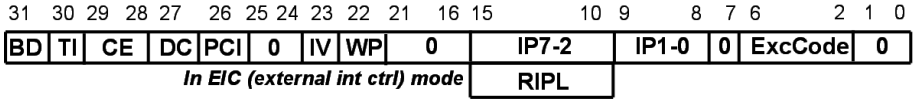

图 3.2: Cause 寄存器的各个域

BD 分支延迟：**EPC** 保存的是异常处理完之后的返回地址。正常情况下，这也指向异常受害指令。

但是如果发生异常的指令是在一条转移指令的延迟槽里，**EPC** 得指向那条转移指令；重新执行转移指令没有什么害处，但如果返回到延迟槽指令本身，转移就不会发生，从而这个异常将破坏被中断的程序。

只要异常发生在延迟槽的指令，**Cause(BD)** 就会置位，**EPC** 就指向分支指令。如果想要分析异常受害指令，只要看看 **Cause(BD)** (如果 **Cause(BD)** == 1，那么该指令位于 **EPC**+4)就知道了。

TI （仅适用于新的 MIPS32/64 CPU ）——该异常由内部定时器中断产生。

CE 协处理器错误：如果异常是由于相应的 **SR(CUx)** 位没有使能某个协处理器格式的指令引起的，那么 **Cause(CE)** 保存这条指令的协处理器号。

DC （仅适用于新的 MIPS32/64 CPU ）——把该位写入 1 可以停止 **Count** 寄存器计数，有时这样做可能是为了省电。

PCI （仅适用于新的 MIPS32/64 CPU ）——CP0 性能计数器溢出而产生此中断。

IV 该位写入 1 表示中断将使用一个特殊的异常入口点，参见第 5.8.5 节。

WP 读出为 1 表示当 CPU 已经处于异常模式时（此时观察点异常被抑制）记住触发的观察点。该位将导致 CPU 一从异常模式返回正常操作就立即触发观察点异常，观察点异常处理程序必须清除该位。

对该位写入 1 中断将使用一个特殊的异常入口点，参见第 5.8.5 节。

IP7–0 待决的中断：给出待发生的中断。**Cause(IP7–2)** 照抄 CPU 硬件的输入信号，**Cause(IP1–0)** （软件中断位）可读可写包含你最近写入的值。当相应的 **SR(IM)** 位（还要受其它禁止中断的条件约束）允许时，这八位中的任意一位活动都会导致一个中断。

Cause(IP) 和 **Cause** 寄存器的其它域有着微妙的不同：它并不告诉你异常触发时发生了什么，而是告诉你现在正在发生什么。

注意对 **Cause(IP7-2)** 的解释当你使用 EIC 中断系统时会有所变化，这在第 5.8.5 节讲到。

ExcCode 这是一个 5 位的编码，告诉你发生了哪种异常，详见表 3.2。这里需要有一个表，但是其中描述的许多条件本书到目前还没有讲到。现在就把它当成一个参考材料；其意义到后面就清楚了。

3.3.3 异常返回地址(EPC)寄存器

这只是一个保存异常返回点的寄存器。导致(或者遭受)异常的指令地址存入 **EPC**，除非 **Cause** 寄存器的 **BD** 位置位了，这种情况下 **EPC** 指向前一条（分支）指令。如果 CPU 是 64 位，么 **EPC** 也是 64 位。

3.3.4 无效虚拟地址(BadVaddr)寄存器

这个寄存器保存引发异常的地址；在发生 MMU 相关的异常时、在用户程序试图访问 kuseg 以外的地址时、或者地址没有正确对齐时，该寄存器被设置。发生其它任何异常后它的值都是未定义的。特别要提请注意的是，发生总线错误时并不设置该寄存器。如果 CPU 是 64 位的，那么 **BadVAddr** 也是 64 位。

3.3.5 Count/Compare 寄存器：CPU 片上定时器

这些寄存器提供一个简单通用的间隔定时器，它连续运行并且可以用编程中断。该中断通常从 CPU 出来然后又连线到 CPU 的中断输入，具体机制与系统有关——参看 **IntCtl** 寄存器以找出具体机制。

Count 是一个连续计数的 32 位递增计数器，以 CPU 的流水线频率的同频、半频或者（很少见）其它分频运行。可以通过读取一个硬件寄存器来找出计数器频率，在第 8.5.12 节讲到。

当 **Count** 计数到最大的 32 位无符号数值时，就溢出而回到零。你可以读取 **Count** 获取当前的“时间”。你也可以随时写入 **Count** ——但是通常不要这么做。

Compare 是一个 32 位的可读写的寄存器。当 **Count** 计数增加到等于 **Compare** 中的值时，中断信号就会升起。中断信号一直保持有效，直至下一次写入 **Compare** 才清除。

要产生一个周期性的中断，中断处理程序应当总是以固定的增量增加 **Compare**（而不是增加 **Count**，因为那样周期就会因为中断延迟而略微延长）。软件需要检查这种可能性——一个迟到的中断响应可能把 **Compare** 设成

表 3.2: ExcCode：异常的不同类型

ExcCode	助记符	描述
0	Int	中断
1	Mod	存储操作时，该页在 TLB 中被标记为只读。
2	TLBL	没有 TLB 转换（读写分别）。也就是 TLB 中没有和程序地址匹配的有效入口。
3	TLBS	当根本没有匹配项（连无效的匹配项都没有）并且 CPU 尚未处于异常模式—— SR(EXL) 置位——即 TLB 失效时，为高效平滑处理这种常见事件而采用的特殊异常入口点。
4 5	AdEL AdES	（取数、取指或者存数时）地址错误：这要么是在用户态试图存取 kuseg 以外的空间，或者是试图从未对齐的地址读取一个双字、字或者半字。
6 7	IBE DBE	总线错误（取指或者读取数据）：外部硬件发出了某种出错信号；具体该怎么做与系统有关。因存储而导致的总线错，只能作为一个为了获取要写入的高速缓存行而执行的高速缓存读操作的结果间接出现。
8	Syscall	执行了一条 syscall 指令。
9	Bp	执行了一条 break 断点指令，由调试程序使用。
10	RI	不认识的（或者非法的）指令码。
11	CpU	试图运行一条协处理器指令，但是在 SR(CU3–0) 中并没有使能相应的协处理器。 具体说，就是当 FPU 可用位 SR(CU1) 没有置位时从浮点操作得到的异常；因而是浮点仿真开始的地方。
12	Ov	自陷形式的整数算术指令（比如说 add 但 addu 不会）导致的溢出。C 语言程序不使用溢出-自陷指令。
13	TRAP	符合了 teq 等条件自陷指令的某一条。
14		目前未用。在有些拥有 L2 高速缓存的老式的 CPU 上，当硬件探测到可能的高速缓存重影时使用这位，在 4.12 节对此有解释。
15	FPE	浮点异常。（在某些很老的 CPU 上，浮点异常以中断形式出现。）
16–17	-	定制的异常类型，与具体实现相关。
18	C2E	来自协处理器 2 的异常（如果有的话，就是对指令集的一种定制的扩展）。
19–21	-	保留给未来扩展使用
22	MDMX	试图运行 MDMX 指令，但是 SR(MX) 位没有置位（很可能该 CPU 没有实现 MDMX）。
23	Watch	load/store 的物理地址匹配了使能的 WatchLo/WatchHi 寄存器。

表 3.2 续

ExcCode	助记符	描述
24	MCheck	机器检查——CPU 监测到了 CPU 控制系统中的灾难性的错误。MIPS 公司的有些核当 TLB 中加载了匹配同一个程序地址的第二个转换项时发出该异常。
25	Thread	线程相关的异常，这在附录 A 中有描述。还有一个寄存器域，VPEControl(EXCPT)，提供有关线程异常的更多细节。
26	DSP	试图运行一个 DSP ASE 指令，但是要么该 CPU 不支持 DSP 指令，要么SR(MMX) 没有设置成使能 DSP。
27–29	-	保留给将来的扩展。
30	CacheErr	在取指、读数、或者高速缓存填充时，核内某个地方发生奇偶校验码/ECC 纠错码错误。这种错误有它们自己的（位于非缓存空间的）异常入口点。事实上，在 Cause(ExcCode) 中从来看不到这个值；但是这个表中的某些编码，包括这个在内，在 EJTAG 调试单元的调试模式下是可见的——参见第 12.1 节，特别是该节关于 Debug 寄存器的说明。
31	-	现在未用，但是历史上用过，跟上面的位 14 差不多。

了一个 **Count** 已经过了的值；典型的做法是在写完 **Compare** 之后重新读取 **Count** 的值。

3.3.6 处理器 ID(PRId) 寄存器

图 3.3 显示了 **PRId** 寄存器的布局，这是一个标识 CPU 类型的只读寄存器。**CPU Id** 应当会各不相同——至少——当指令集或者控制寄存器定义发生变化时，**CPU Id** 的值要改变。“Revision”完全取决于制造厂家，只是用来帮助 CPU 厂家跟踪芯片版本，用做其它任何用途都是不可靠的。

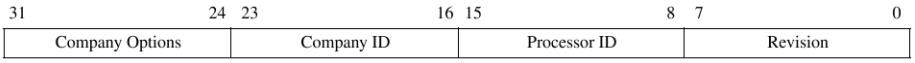

图 3.3: PRId 寄存器域

Company Id 域——可以从 MIPS 公司得到该域的取值——相对较新，历史上以往的 CPU 都将其设置为零。**Company Options** 只在你的 CPU 手册中有定义。

表 3.3 中列出了我们所知道的一些 **CPU Id** 的设置。

如果您想打印出这些值，习惯上写成“x.y”的形式，其中 x 和 y 分别为 **CPU ID** 和 **Revision** 的十进制值。尽量避免依赖这个寄存器的内容来确定某些参数（比如高速缓存大小，速度等等）或者确定某项特性的有无。有些特性在

某个 **Config** 寄存器中有标准的编码来确定，或者也可以写一段代码序列来探测单个特性是否存在。

表 3.3: PRId(Imp) 中的 MIPS CPU 实现号

Id	CPU 类型	Id	CPU 类型
1	R2000, R3000	128	MIPS 4KC
2	IDT R305x family	129	MIPS 5KC
3	R6000	130	MIPS 20KC
4	R4000, R4400	131	MIPS 4KMP
5	Early LSI LOgic 32-bit CPUs	132	MIPS 4KEc
6	R6000A	133	MIPS 4KEmp
7	IDT R3041	134	MIPS 4KSc
9	R10000	135	M4K
10	NEC Vr4200	136	MIPS 25Kf
11	NEC Vr4300	137	MIPS 5KE
12	NEC Vr41xx family	144	MIPS 4KEc(MIPS32R2 兼容)
16	R8000	145	MIPS 4KEmp(MIPS32R2 兼容)
32	R4600	146	MIPS 4KSd
33	IDT R4700	147	MIPS 24K
34	Toshiba R3900 family	149	MIPS 34K
35	R5000	150	MIPS 24KE
40	QED RM5230, RM5260		

3.3.7 Config 寄存器：CPU 资源信息和配置

MIPS32/64 标准定义了四个标准寄存器给初始化软件使用：Config 和 Config1–3。大多数寄存器域是只读的，软件一审问就会交待出 CPU 硬件的相关信息；但是（特别是最初的 **Config** 寄存器——由于历史的原因而没有叫做 **Config0**）也有一些可写的域，用来选择对一个具体系统只能选其一的 CPU 选项。

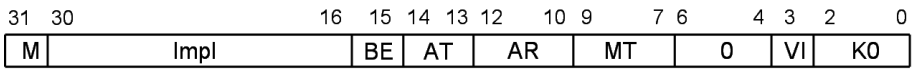

图 3.4: MIPS32/64 Config 寄存器的各个域

所有 MIPS32/64 兼容的 CPU 都有 Config 寄存器，如图 3.4 所示，有以下的这些域：

M 接续位——读出值为 1 说明至少还有一个配置寄存器（即 Config1 ）可用。

Impl 依赖于具体实现的配置。大多数 CPU 在这里有一些定制的域，既可以读出信息也可以写入系统设置。请参看具体 CPU 的手册。

BE 1 代表大尾端，0 代表小尾端。

AT 是MIPS32 还是 MIPS64？编码如下：

0 MIPS32

1 MIPS64 指令集但是 MIPS32 地址空间映像

2 MIPS64 指令集全地址空间

AR 体系结构版本号：

0 MIPS32/MIPS64 第一版

1 MIPS32/MIPS64 第二版

本书是基于 MIPS32/MIPS64 第二版。

MT MMU 类型：

0 没有（None）

1 MIPS32/64 标准的 TLB

2 BAT 类型

3 MIPS32 标准的 FMT 固定映射

BAT 类型是由于对一些老式 CPU 的历史兼容性原因而保留的，不大可能还碰得到。

VI 置 1 代表 L1 一级指令高速缓存 I-cache 以虚拟地址（即程序地址）索引和标记。虚拟的 I-cache 要求操作系统作特殊的处理/对待。

K0 可写的域，用来决定固定的 kseg0 区是否经过高速缓存。如果要经过高速缓存，其确切行为如何？该域的编码和在 **EntryLo0-1(C)** 中见到的 TLB 项的高速缓存选择域一样，在图 6.3 的注释中有说明。

图 3.5 有如下的数据域：

Config1(M) 继续位，为 1 代表还实现了 **Config2**。

Config1(MMUSize) TLB 数组的大小减一。（该数组有 MMUSize+1 项）。

L1 I-cache, L1 D-cache 对每个高速缓存，报告以下的三个值：

S 高速缓存可直接索引的位置数为 64×2^S。乘以相联数得到高速缓存的数据行数。

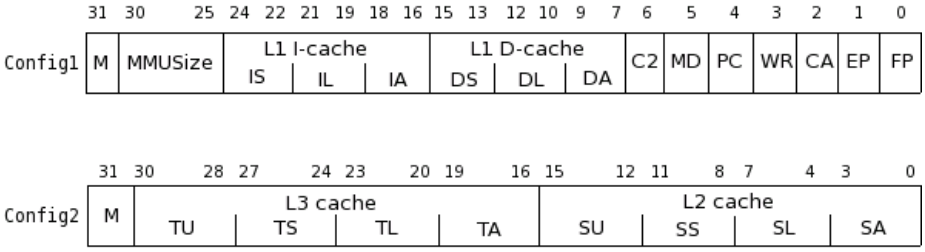

图 3.5: MIPS32/64 Config1–2 寄存器的各个域

L 零代表根本没有高速缓存；否则告诉你高速缓存的行大小是 2×2^L 字节。

A 相联度——该高速缓存为 $(A+1)$ 路组相联。

所以如果(IS, IL, IA)为(2，4，3)，就是说高速缓存有 256 组/路，32 字节/行，并且是四路组相联：那就是 32-KB 字节的高速缓存。

Config1(C2) 为 1 则表明有协处理器 2（很可能是某些专用的协处理器）。

Config1(MD) 为 1 则表明在浮点部件里实现了老的 MDMX ASE（参见附录B.3）。

Config1(PC) 表明至少实现了一个性能计数器，参阅第12.2 节。

Config1(WR) 为 1 表明 CPU 至少有一个观察点寄存器，参见第12.2 节。

Config1(CA) 当 MIPS16e 压缩码指令集可用时为 1，参见第12.1 节。

Config1(EP) 如果提供了 EJTAG 调试单元则为 1，参见第12.1 节。

Config1(FP) 附加了一个浮点单元。

Config2(M) 继续位，1 代表实现了 **Config3**。

Config2(TU) 与 L3 高速缓存相关的具体实现用的位，有可能可写。

Config2(TS, TL, TA) L3 高速缓存大小和形状——和 **Config1(IS, IL, IA)** 相同编码。

Config2(SU) L2 高速缓存相关的具体实现用的位，有可能可写。

Config2(SS, SL, SA) 二级高速缓存大小和形状——和 **Config1(IS, IL, IA)** 相同编码。

图 3.6 中包括：

Config3(M) 继续位，（极可能）为 0 代表没有 **Config4**。

	31	30 … 11	10	9	8	7	6	5	4	3	2	1	0
Config3	M	0	DSPP	0		LPA	VEIC	VInt	SP	0	MT	SM	TL

图 3.6: MIPS32/64 Config3 寄存器的各个域

LPA 当支持大的物理地址(LPA)时为 1，此时允许物理地址的范围超过 2^{36} 字节。当有 LPA 支持的时候，还会有一个额外的名叫 **PageGrain** 的 CP0 寄存器，同时 **EntryLo0-1** 和 **EntryHi** 中的域的布局也会变化。

MIPS32 的核（到目前为止）没有实现 LPA 的，所以本书不讨论。需要时请参考厂商的手册。

DSPP 如果实现了 MIPS DSP 扩展则该位为 1，如附录 B.2 节所述。

VEIC 只读位，表明是否有 EIC 兼容的中断控制器；参见第 5.8.5 节。

注意，中断控制器并不属于核的一部分，所以典型的做法是由系统设计者通过布线连到 CPU 或者 CPU 核。

VInt 如果 CPU 可以处理向量化的中断则读出为 1。

SP 如果 CPU 支持小于 4 K 的页面大小则读出为 1。大多数通用的 MIPS CPU 不支持。

MT 如果 CPU 实现了多线程则读出为 1，附录 A 有关于 MIPS MT 扩展的资料。

SM 如果 CPU 可以处理来自“SmartMIPS”ASE 的指令则读出为 1。该扩展对在慢速 CPU 上的加密例程提供了帮助，大多数定位在为智能卡建造的 CPU。

TL 读出时为 1 表示你的核可以记录和输出指令跟踪信息。指令跟踪是 EJTAG 调试单元的一种高级的可选特性，在 12.1 节有描述。

3.3.8 EBase 和 IntCtl：中断和异常设置

这些寄存器（在 MIPS32/64 第二版中新增的）让你能够控制那里新增的中断能力。

加上了 EBase 允许你重新定位一个 CPU 的所有异常入口点；其主要是用于给共享内存的多处理器系统，这样不同的 CPU 就不必用同样的异常处理程序了。

图 3.7 中给出了 EBase 寄存器的所有域：

1，0 只读位作为基址的前缀位以确保异常向量最后落入 kseg0 区，习惯上是给操作系统代码用的。

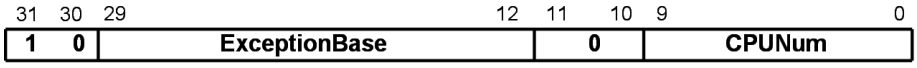

图 3.7: EBase 寄存器的布局

ExceptionBase 是异常向量的基址，可以调整到任意 4KB 地址的边界。参见表 5.1，其中可以查到调整后的所有异常入口点在什么地方。

这意味着任何一个 CPU（甚至包括多线程 CPU 上的“虚拟” CPU）都可以有自己独立的异常处理程序。

CPUNum 一个数字用以区分这个 CPU 和同一个多处理器系统其它的 CPU。该域的内容极为可能是由你的系统设计人员通过硬件布线确定的。

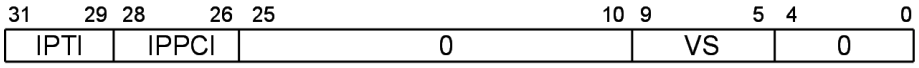

图 3.8: IntCtl 寄存器的布局

然后是寄存器 **IntCtl**，如图 3.8 所示：

IPTL, IPPCI 都是只读域，告诉你定时器和性能计数器中断（CPU 内部产生的）是怎样连线到你的系统中去的。这在非向量化以及简单向量化（VI）的中断模式中有用。

每个域都是 3 位的二进制数，标识和哪个 CPU 中断输入被内部的定时器中断（IPTI）或性能计数器溢出中断（IPPCI）共享。

哪个中断通过给出 **Cause(IPx)** 位号指定，从那里可以看到中断。因为 **Cause(IP0–1)** 是软中断位，不连到任何输入，所以 **IntCtl(IPTI)** 和 **IntCtl(IPPCI)** 的合法值在 2 到 7 之间。

定时器和性能计数器中断被引出到 CPU 接口的外部，然后在那里又沿着其中一个中断信号送回。所以这个信号在 CPU 内部不能确定，而是由系统设计人员提供的。

VS 可写的，可让你用软件控制向量之间的间隔：相连的两个向量入口的间距为 **IntCtl(VS)** ×32 字节。只有 1、2、4、8 和 16 这些值（分别给出 32、64、128、256 和 512 字节的间距）能用。零值真的给出零间距，这时所有的中断就都到同一个地址。

3.3.9 SRSCtl 和 SRSMap：影子寄存器的设置

影子寄存器是 MIPS32/64 第二版新增的特性。CPU 装备了一个或者多个额外的通用寄存器组，并且在异常发生时——特别是中断发生时——切换到另外的组。

31 30	29 26	25 22	21 18	17 16	15 12	11 10	9 6	5 4	3 0
0	HSS	0	EICSS	0	ESS	0	PSS	0	CSS

图 3.9: SRSCtl 寄存器的布局

在图 3.9 所示的 **SRSCtl** 中：

HSS 在本 VPE/CPU 上可用的寄存器集的最高编号（即可用的寄存器集的个数减一）。

在多线程 CPU 上该域可能被分配影子寄存器集的软件修改。但多数情况下，该域是只读的。

CSS 当前使用的寄存器集。在这里是只读的；异常时自动设置，**eret** 指令返回时用 **SRSCtl(PSS)** 中的值取代。

ESS 该域可写，由软件选择的可用于“所有其它”异常的寄存器集；所谓“其它”是指除了 VI 或者 EIC 模式（二者都有自己选择寄存器集的特殊方法）的中断之外。该域取非零值的情况极为罕见。

PSS “前一个”寄存器集，随后的下一条 **eret** 指令会用到。

你可以用 **rdpgpr** 和 **wrpgpr** 获取该组寄存器中的值。

SRSCtl(PSS) 域可写，允许操作系统在新的寄存器集中快速处理代码；加载这个值然后执行一条 eret。

EICSS 在 EIC 模式（参见 5.8.5 节），一个外部中断控制器每次请求中断（非零的 IPL）时提议一个影子寄存器组号。当 CPU 接受一个中断，外部提供的组号决定下一个组号，并且该组号直到下一个中断前是可见的。

只是提醒一下：并不是所有异常都会更新 **SRSCtl(PSS)** 和 **SRSCtl(CSS)**，更新仅限于那些向 **EPC** 写入新的返回地址的异常（换句话说，异常级别位 **SR(EXL)** 从零变为一的场合）。不写入 **EPC** 的异常包括：

- **SR(EXL)** 已经置位时发生的异常
- 高速缓存出错异常，此时返回地址加载到 **ErrorEPC**
- EJTAG 调试异常，此时返回地址加载到 **DEPC**

3.3.10 连锁加载地址（LLAddr）寄存器

该寄存器保存着上次/最近运行过的连锁加载操作的物理地址，保存该值是为了监控可能导致将来条件存储失败的访问；参见第 5.8.4 节。软件对 **LLAddr** 的访问仅用于诊断。

3.4 CP0 遇险——提防落入陷阱

因为 CPU 是流水线化的，CP0 操作的效果可能直到 CP0 指令的后期阶段才能到达其目标——即便此时可能还要几个时钟传播开来到达硬件。但是即使在我们的 CP0 指令执行到流水线后期之前，其它指令可能已经取出并开始了。我们怎样才能保证这些指令能按照我们变化后新的 CP0 执行呢？

理论上说，由硬件工程师找出每个可能的交互并互锁，从而造出一个软件人员根本不需关心这一问题的 CPU。SGI 的 R10000 就实现了这一理想。

但是 CP0 操作往往隐晦难懂，许多操作很少用到，且都是在本来就必须信任的 OS 软件的控制之下。这样在 MIPS 体系结构和 CPU 设计人员看来，把其中一些难题推到软件人员的肩膀上是合理的。

历史上，软件人员要通过分析程序的流程，查出可能出问题的地方并进行修正。他们手上的武器就是 **nop** 指令，在变化传播过程中什么也不干以确保安全；以及分支/ 转移指令，它们会“作废”顺序取出的指令序列，强制重新取指和重新执行。如果你看一下兼容 2003 年版（第二版）的 MIPS32/64 标准之前的 MIPS CPU 手册的封底，通常都会找到一张有关 CP0 遇险的表，表中详细描述了为了保证后继指令看到已经生效的各种变化所需要花费的传播时间。

即使是早期的只有简单流水线的 CPU，做到这点都有点挑战性。随着流水线的加长和更为复杂，这对于可移植的软件来说真成了一个麻烦。一旦有 CPU 可以并行执行指令（简单的双发射甚至乱序执行），计算需要多少条 nop 指令就变得很困难了。一度时期，我们甚至都有了 **ssnop**(superscalar no-op) 指令，单独发射该指令就能保证耗掉一个时钟周期。

但是 MIPS32/64 CPU 现在有了一个更为切合实际的方法：遇险防护指令。这些是特殊的指令，只要放在需要的地方，就能延迟后继指令直到先前的 CP0 指令的所有效果传播到位。

MIPS32/64 根据依赖指令受影响的是哪个阶段区分两种不同的 CP0 遇险。执行遇险是指那些依赖的指令直到读取一个 CP0 寄存器值的时候才受到影响的情形。指令遇险是指那些依赖的指令在其最早期就受到影响的情形——最坏的情形是，从内存或者高速缓存取指开始就已经受到影响。

另一个有意思的区别是那些只影响其它 CP0 指令（必然是操作系统的一部分）的遇险和那些（我们可以称其为用户态遇险）可能影响普通指令的遇险。

3.4.1 遇险防护指令

在我们开始之前，注意到任何异常都会消除所有遇险（所以不会因为异常处理程序开始时某个操作尚未完成而导致出错），对于 **eret** 也是一样——操作系统不管做任何事情都不会把麻烦带回给用户程序。

有三个明确的遇险防护指令。你可以用 **ehb** 消除执行遇险——老式的 CPU 把它当成是一个 no-op。指令遇险通过特殊的寄存器转移指令 **jr.hb** 和 **jalr.hb**

来消除，最常见的用法是分别替换正常的子程序返回和调用指令。

MIPS 架构师在选择特殊的寄存器转移指令时很有先见之明，所选指令在老式的 CPU 上恰好译码成 **jr**、**jalr** 指令。在这些 CPU 上，这种指令清空 CPU 流水线（寄存器转移指令本质上是“不可预测的”），并对不符合后来的 MIPS32/64 规范的 CPU 在大多数情况下会提供必要的延迟。

3.4.2 指令遇险和用户遇险

这种现象发生的典型情景是当我们对 CP0 状态（某个寄存器、某个 TLB 表项、或者高速缓存某行）进行的修改会影响到我们正常取指的方式（或者在少数情况下，影响load/store 指令访问内存的方式）。这种遇险必须在我们返回到任何“非控制的”的代码之前加以解决。

在这些情况下，你必须将遇险防护指令置于改变状态的 CP0 操作之后。最常见的情形是把它放在紧挨其后的位置——但是你也许会有其他工作你知道可以在遇险操作的效果传播过程中安全执行。但这得由你自己的脑袋决定。

这种类型的遇险包括：

TLB 表项的改变	⇒	在受影响的页面上取指、存储和读取操作数
EntryHi(ASID 域) 的改变	⇒	非全局映像的取指、存储和读取操作数
改变到 **ERL** 模式	⇒	从 kuseg 区的取指、存储和读取操作数
修改高速缓存行的 **cache** 指令	⇒	在受影响的高速缓存行上的取指、存储和读取操作数
观察点寄存器的改变	⇒	在相匹配的地址上的取指、存储和读取操作数
影子寄存器设置的修改	⇒	任何使用通用寄存器的操作（执行遇险）
禁止中断的 CP0 寄存器修改	⇒	依然能够被中断的指令（执行遇险）

大多数这些属于指令遇险——当没有 **eret** 为这些指令提供足够的防护的时候，你应当用一条 **jr.hb** 或者 **jalr.hb** 指令。执行遇险可用一条 **ehb** 指令来消除。

3.4.3 在 CP0 指令之间的防护

任何 **mfc0** 指令都显式依赖于一个 CP0 寄存器中的值，但是因为所有的 TLB 信息都要通过寄存器进行，**tlbwi**、**tlbwr** 和 **tlbr** 也都一样。类似的，那些从 CP0 寄存器读出数据的 **cache** 指令也有依赖。

tlbp 通过其 **ASID** 域而依赖于 **EntryHi**，这一点就没有那么明显。

所有这些都属于执行遇险，通过在需要读取信息的 CP0 指令之前放置一条 **ehb** 指令就可以保证安全。如果你有机会把它放到早几条指令的位置上去，那么越早越好（有些 CPU 在一条 **ehb** 之后的几个时钟周期内可能会禁止所有的 CP0 指令）。

第 4 章　MIPS 的高速缓存工作机制

没有高速缓存(cache)的 MIPS CPU 不是真正的 RISC。这样说也许有失公允；理论上说，你可以为了特殊用途而设计一个没有高速缓存的 MIPS CPU，该 CPU 只包含一个能在固定个数（最好就是一个）的流水线周期访问到的紧耦合的小容量内存。但实际的 MIPS CPU 都含有紧密耦合进流水线的高速缓存。

这一章将介绍 MIPS 的高速缓存工作机制，以及软件应该怎么做才能可靠有效的利用高速缓存。复位时，与高速缓存有关的状态几乎都是不确定的，所以引导软件必须非常小心，在使用之前要正确初始化高速缓存。当确定高速缓存大小的时候，有些线索和技巧很有用（假定高速缓存的大小事先已知是一个不好的软件习惯）。对于诊断程序员，我们将讨论怎样测试高速缓存和检测特定的缓存项。

有些实时应用程序的程序员，可能想要能够精确控制哪些部分在高速缓存中运行。我们也将讨论这样做的一些窍门，但是我对使用这些窍门是否明智表示怀疑。

发展过程中也有过一些与本章介绍的相竞争的高速缓存管理机制。早期的 32 位 MIPS 处理器，高速缓存的管理采用另一种方案：先让高速缓存进入一种特殊状态，然后通过普通读写操作的副作用对高速缓存进行初始化或者作废其内容。但是我们这里不会详细讲这个，即使不完全兼容 MIPS32/64 标准的 CPU，一般也采用很接近于我们在下文中描述的机制。

4.1　高速缓存及其管理

高速缓存的工作就是将内存中最近读写的一部分数据保留一个备份，使这些数据能快速存取并返回给 CPU。对于一级(L1)高速缓存，必须在固定的时间内完成存取以保证流水线的连续运行。

MIPS CPU 指令和数据各自有相应的一级高速缓存（分别称为 I-cache 和 D-cache），这样读取一条指令和存取一个数据的操作就能同时进行。

带有高速缓存的传统 CPU 家族（比如 x86）要保证能够兼容为没有高速缓

存的 CPU 写的代码。当代的 x86 芯片拥有精心设计的硬件可以保证软件根本没有必要了解高速缓存（假如你正在装一台要跑 MS-DOS 的机器，这种硬件对保持向后兼容性是必不可少的）。

但 MIPS 机器因为总是拥有高速缓存，所以象上面那么智能化的高速缓存就不是绝对必要。高速缓存对应用软件必须保持透明，唯一的变化只是运行速度的增加。但是 MIPS CPU 因为总有高速缓存硬件，就没有人试图让高速缓存对系统程序或者驱动程序也保持透明——安装高速缓存硬件的目的是为了让 CPU 跑得更快，而不是为了帮助系统程序员。当然 Unix 之类的操作系统对应用程序完全隐藏高速缓存，但是更轻量级的操作系统可能只隐藏高速缓存操作的细节，你可能仍然需要知道在什么时候调用适当的例程。

4.2 高速缓存是怎样工作的

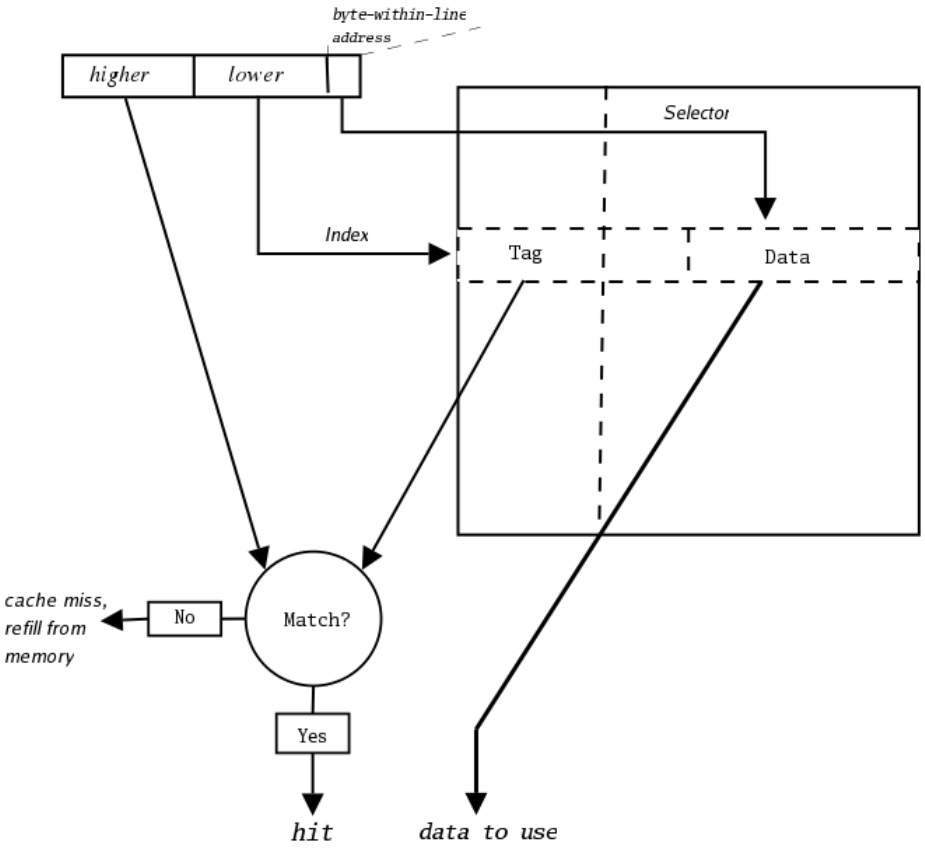

图 4.1: 直接映射的高速缓存

从概念上讲，高速缓存是一个相联存储器（associative memory），数据存入后可以用任意数据模式作为关键字来查找。在高速缓存中，关键字是完整的内

存地址。提供相同的关键字给相联存储器，你将得到相同的数据。一个真正的相联存储器可以接受使用任意关键字的新数据项，除非它已经满了。然而，由于给出的关键字必须和每个存储的关键字同时比较，任何合理大小的真正全相联存储器要么消耗资源太多，要么速度太慢，甚至两者都有。

怎样才能设计一个快速高效有用的高速缓存？图 4.1 展示的高速缓存基本布局属于一种最简单的高速缓存，即 1992 年以前的大多数 MIPS CPU 使用的直接映射（direct-mapped）高速缓存。

直接映射高速缓存使用一块简单的高速存储器（*缓存栈*），通过足够多的地址低位索引可以扩展容量。缓存栈中的每一行包含一个或多个字的数据和一个记录该数据内存地址的标签（tag）域。

每次读操作访问高速缓存行时，其标签域与内存地址的高位做比较；如果匹配，我们就知道正确找到了数据，称为“命中（hit）”了高速缓存。当该行包含多个字的数据时，通过地址的最低几位来选择出对应的那个字的数据。

如果没有标签相匹配，那么没有找到数据，需要从内存中读入数据，然后复制到高速缓存中。高速缓存相应位置处原先的数据就简单丢弃，CPU 需要时，再从内存中重新读取。

这样的直接映射高速缓存有个性质，就是对于任意给定的内存地址，在高速缓存中只有唯一的行可以保存其数据。[1] 这样有好处也有坏处。好处就是这样简单的结构可以让整个 CPU 跑得更快。但简单也有其不利的一面：如果你的程序要不停地交替使用两个数据，而它们刚好（可能是它们地址的低位刚好在一起）共享高速缓存中同一个位置，这样这两个数据就会不停的将对方替换出高速缓存，效率将急剧下降。

而真正的相联存储器不会遇到这样翻来覆去的来回折腾，但是太慢太贵。

折衷的办法就是使用两路组相联的高速缓存——其实就是两个并行的直接映射高速缓存，同时在两个当中查找内存地址，如图 4.2 所示。

这时对应一个地址将有两次命中机会。四路组相联的高速缓存（等效于有四个直接映射的子高速缓存并列）在片上高速缓存的设计中也很常见。

在多路相联的高速缓存中，解决高速缓存未命中时有不止一个高速缓存位置可选。这样要丢弃替换的高速缓存行的数据也有不止一个可以选择。理想的解决方法是跟踪对高速缓存行的访问，挑选出“最近最少使用（LRU）”的高速缓存行进行替换，但是维护严格的 LRU 次顺序意味着每次读取高速缓存时，都要更新每个高速缓存行的 LRU 位。而且对于超过四路组相联的高速缓存来说，维护严格的 LRU 信息变得不切实际。实际的高速缓存常常使用象“最近最少填充”这样折衷的算法来选择要替换的高速缓存行。

当然这是有代价的。比起直接映射的高速缓存来，组相联的高速缓存在缓存芯片和控制器间需要更多的总线连接。这就是说，太大的难以集成在单个芯

[1] 在全相联存储器中，与给定内存地址（即关键字）相联的数据可以存放到任意位置；直接映射高速缓存距离内容寻址远得不能再远了。

片上的高速缓存做成直接映射的要容易得多。更为微妙的地方在于，由于直接映射的高速缓存对于需要的数据只有唯一的候选位置，有可能让 CPU 比标签检查提前运行（只要 CPU 不根据该数据执行不可回撤的操作）。简单和提前运行等效于更快的时钟频率。

当运行一段时间后高速缓存会被装满，所以获取新的内存数据时通常就要丢弃前面缓存的一些数据。如果你知道这些数据已经安全写入了内存，你可以直接丢弃高速缓存的内容；但如果高速缓存中的数据比内存的要新，你就需要先把这些数据写到内存。

这就带给我们一个新问题：高速缓存怎样处理写操作。

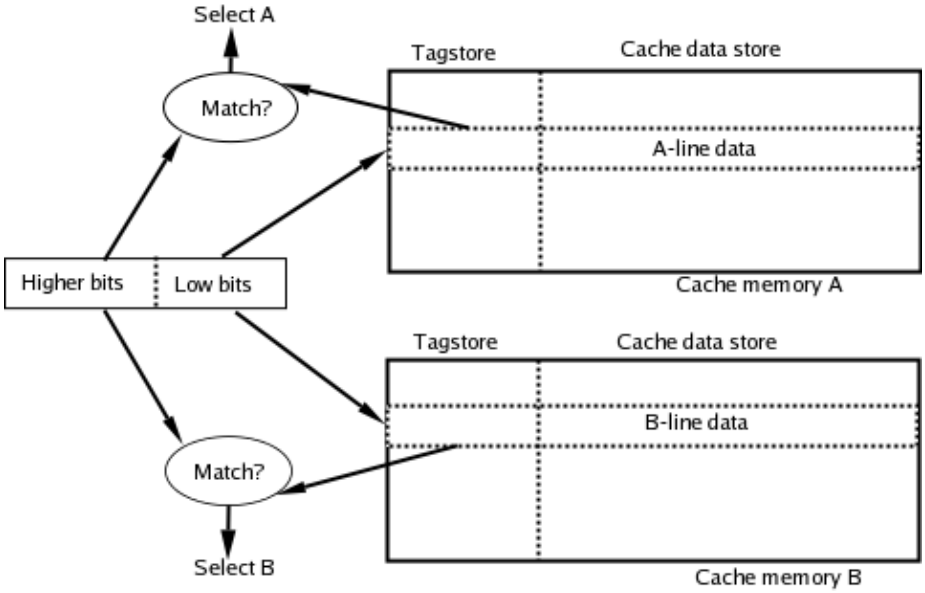

图 4.2: 两路组相联高速缓存

4.3 早期 MIPS CPU 的透写高速缓存

CPU 并不是仅仅读数据（上面的讨论似乎假定只读），它也要写数据。由于高速缓存只是将主存中的一部分数据做一个局部备份，所以处理 CPU 的写操作的一个显而易见的方法就是采用被称之为透写高速缓存的方法。

在透写式高速缓存中，CPU 的数据总是写到主存中去；如果对应主存位置的数据在高速缓存中有一个备份，那么这个备份也要更新。如果我们永远这样做，那么高速缓存中的任何数据将和主存中的保持一致，所以我们随时可以丢弃某个高速缓存行的内容，而不丢失任何数据。

要是我们等待主存透写操作完成，就会急剧降低处理器的运行速度，不过这个问题可以纠正。在主存控制器进行准备和写入数据的同时，将要写入主存

的数据及其地址先保存到一个队列。然后由主存控制器自己取得这些数据并完成写操作。这个临时保存写操作的地方被组织成一个先入先出的（FIFO)的存储区，称为*写缓冲器*（*write buffer*）。

早期的 MIPS CPU 有一个直接映射的透写高速缓存和一个写缓冲器。只要主存系统能够很好的消化这些 CPU 以平均速率产生的写操作，以这种方式运行特定程序就工作得很好。

但 CPU 速度的增长比存储器快得多。当 32 位的 MIPS 让位给 64 位的 R4000 前后的那段时间，MIPS CPU 的速度就已经超过内存系统可以合理消化所有写操作的临界点了。[2]

4.4 MIPS CPU 的回写高速缓存

尽管早期的 MIPS CPU 使用简单的透写高速缓存，后来的 CPU 由于太快而不适用这种方法——它们的写操作会拥塞存储器系统，让速度慢得象蜗牛爬行。

解决的方法就是把要写的数据保留在高速缓存中。要写的数据只写到高速缓存中，并且对应的那条缓存行要做一个标记，保证我们以后不会忘记把它写回到主存中（一个需要写回到主存的高速缓存行，称之为 dirty 或者 modified）。[3]

回写还可以分成几种不同的子方式。如果当前高速缓存没有要写的地址所对应的数据，我们可以直接把数据写到主存中而不管高速缓存，也可以用特殊的方式把数据先读入高速缓存然后再写——这种方式被称之为写分配（write allocate）。写分配未必总是最好的做法：对写给供 I/O 设备读取（ CPU 不会再次读写）的数据，理想的做法是不经过高速缓存。但这只有靠程序员告诉操作系统才能做到。

除了极为低档的之外，现代的 MIPS CPU 都集成了行大小为 16 或 32 字节的片上回写式高速缓存。

R4000 和后来的一些 CPU 用在 SGI 和其它公司的大型计算机服务器上。它们的高速缓存设计选择还要受到多处理器系统需求的影响。在第 15.3 节简短介绍了典型的多处理器系统。

4.5 高速缓存设计的一些其它考虑

在 1980 和 1990 年代针对高速缓存该怎样设计做了很多工作和探索。下面介绍其它一些要考虑的地方：

[2]关于程序的一个很粗略的经验数据表明，大约每十条指令就有一个写操作，所以透写的解决方法一直到存储器周期达到大约 5–7 条指令周期时仍有效。那时 DRAM 的一个周期大约为 180 ns，这种简单的解决方法在大约 30–40 MHz 时就失效了。

[3]你可能会问，既然我将来迟早要写回主存的，当然也可以现在就写回了？但是实际上程序经常对同一小块主存连续写好多次，回写高速缓存允许把许多单个的写操作合并为对主存的一次写入。

- 物理寻址/虚拟寻址（Physically addressed/virtually addressed）：CPU 在运行成熟的操作系统时，程序中的数据和指令地址（程序地址或虚拟地址）会被转换成系统存储器的物理地址。

  纯粹工作在物理地址方式下的高速缓存很容易管理（我们将在后面解释为什么）。但原始的程序地址（即虚拟地址）可以更早地开始高速缓存匹配，让系统跑得稍微快一点。

  那么程序地址有什么问题呢？它们不是唯一的；运行在一个 CPU 上不同的地址空间内的许多不同的程序使用的不同数据可能会共享同样的程序地址。当我们在不同的地址空间切换时，可以每次都重新初始化整个高速缓存；这种方式多年就用过，针对很小的高速缓存可能是一种合理的做法。但是对于大的高速缓存，这种方式效率低得可笑；而且要在高速缓存标签中包含一个地址空间标识域以防止混淆。

  许多 MIPS CPU 都采用程序（虚拟）地址为一级高速缓存提供快速的索引。但是却采用物理地址标记每一个高速缓存行，而不是采用程序地址加地址空间标识符。对应高速缓存行的物理地址是唯一的而且效率更高，因为这样的设计允许 CPU 在查找高速缓存的同时进行程序地址到物理地址的转换。

  关于程序地址还有一个更加微妙的问题，采用物理地址标签也不能解决。同样的物理地址在不同的任务中可能被描述成不同的程序地址。这可能导致同样的物理地址被存入到高速缓存两个不同的数据项（因为使用不同的虚拟地址选择了不同的高速缓存索引）。许多 MIPS CPU 没有硬件来检测或者避免这种高速缓存重影（*cache alias*）而把问题留给操作系统的内存管理程序来解决，详细情况请参阅第 4.12 节。

- 高速缓存行大小的选择：高速缓存行的大小指对应每一个标签可以存贮多少字的数据。早期 MIPS 的高速缓存一个标签只能对应一个字，但一个标签能对应多个字的数据通常更好，尤其是在内存系统支持快速的突发读取时（大多数都支持）。现代的 MIPS 高速缓存趋向于使用四个字或八个字大小的行，但是大容量的二级和三级高速缓存可能使用更大的行。

  当发生高速缓存未命中时，要从内存中重新读取整行的数据。

- 分开/统一（Split/unified）：MIPS 的一级高速缓存总是分成 I-cache 和 D-cache，选择哪一个完全取决于功能，取指令时察看 I-cache，读写数据时察看 D-cache。（顺便说一下，这等于说，如果你想执行 CPU 刚刚拷贝到内存的代码，你必须不仅仅要把这些新指令从 D-cache 写回到内存，而且还要作废当前的 I-cache，以保证你执行的确实是新的代码）。

  但是二级高速缓存很少做这种划分——这样做复杂、昂贵而且性能不佳。

4.6 管理高速缓存

我希望你还记得在 2.8 节讲过一个 MIPS CPU 有两个固定的 512M 大小的物理内存窗口，一个进行高速缓存（“kseg0”）另一个不进行高速缓存（“kseg1”）。典型的做法是，操作系统代码运行在 kseg0，并且用 kseg1 生成要求不经高速缓存的引用。

高于 512M 的物理地址在这里是不可访问的：64-位 CPU 可以通过另外一个窗口直接访问，或者你可以设置 TLB(内存管理/转换硬件）进行地址映射。每个 TLB 表项都有标志表明访问是否经过高速缓存。

高速缓存硬件在系统软件的帮助下，必须保证任何应用程序得到的数据和在没有高速缓存的系统下是同样的，而且任意直接存储器访问（DMA）I/O 控制器（直接从存储器中取得数据）得到的数据就是该程序认为自己写入的数据。

我们此前曾经说过在 PC 和一些其它集成系统中，系统软件的帮助常常没有必要；为了能够让高速缓存真正透明，值得在硬件上花这个钱、芯片面积、额外的周期的代价。这种硬件管理的高速缓存称为一致的

CPU 高速缓存阵列的内容在上电后通常都是随机的。引导软件负责初始化高速缓存；这可能是一个非常复杂的过程，下面给了几点建议。但是一旦 CPU 已经启动运行起来了，只有三种情况才需要 CPU 干预高速缓存：

- 在 DMA 设备从内存取数据之前：如果一个设备从主存中取出数据，它取得正确的数据就至关重要。如果数据高速缓存是回写式的，并且程序最近已经写了一些数据，那么很可能其中一些新正确的数据还保留在 D-cache 高速缓存中而没有写回到主存中去。CPU 当然无法发现这个问题：CPU 查看主存地址时，会从高速缓存中得到新的正确的数据。

  所以在 DMA 设备从内存中开始读数据前，如果被读取的区域还有数据留在 D-cache 中，就必须被写回到主存。

- DMA 设备写数据到内存：如果一个设备要将数据存储到内存中，对应的于写入内存地址的高速缓存行的内容都必须作废，这点很重要；否则，CPU 读这些地址时将得到高速缓存中过时的数据。相应的高速缓存数据必须在 CPU DMA 输入流的数据之前作废，但更常见的做法是在 DMA 开始之前就作废。

- 指令写入：当 CPU 自己写指令到内存中，后面再执行这些指令时，你首先必须保证这些指令会被回写到内存中，其次要保证对应的 I-cache 位置上作废。MIPS CPU 的 D-cache 和 I-cache 之间是没有联系的。

  在大多数现代的 MIPS CPU 中，**synci** 指令完成一切必要的操作以便让你刚刚写入内存的指令能够用于执行——它还是一条用户特权级的指令。

为什么不通过硬件来管理高速缓存?

硬件来管理的高速缓存通被称为“一致的”(或者更正式一点,“侦听”)。当另一个 CPU 或者是 DMA 设备访问内存时,就会告知高速缓存控制逻辑。有个连接到共享总线上的 CPU,这个事就很容易做到;地址总线包含了所需要的大多数信息。即使在 CPU 不用总线的时候,高速缓存控制逻辑也在观察(侦听)地址总线,挑出有关的总线周期。它通过查找自己的高速缓存来看看自己是否有待访问地址的数据拷贝。

如果有人对高速缓存内的地址写数据,控制器可以观察到数据并且更新自己的拷贝,但更可能是作废自己已经过时的老数据。如果有人读取在高速缓存中更新的数据,控制器可能干预总线,告诉内存控制器现在有了更新的数据版本。

这种做法的一个主要问题是只适用于设计成那种方式工作的系统。并非所有系统都有一个所有事务都出现在上面的单一总线,从外部购置的 I/O 控制器不大可能正好遵守正确的协议。

还有要做很多侦听。CPU 用到的大多数地址属于该 CPU 的专用区;永远不会被其它 CPU 和设备读写。我们不想在高速缓存的硬件中花太多心思,为了只在个别时候才用到的功能而给每个高速缓存位置和总线周期都加上额外的复杂度。人们容易认为硬件高速缓存控制肯定比软件快,但事实并不一定如此。一个侦听的高速缓存控制器必须在每个外部周期都要察看缓存标签,这可能让 CPU 没法使用高速缓存而变慢;侦听的高速缓存控制器通常解决这个问题的办法就是保持高速缓存标签的两份拷贝(一个 CPU 专用的版本和一个公用的版本)。软件管理可以在单个快速循环中同时操作多个高速缓存块。硬件管理需要交替回写和作废操作让 CPU 以 I/O 速度访问,这通常意味着更多的仲裁开销。

历史上,MIPS 设计者采取了激进的 RISC 立场: MIPS CPU 要么完全没有高速缓存管理硬件,要么就全都有(为多处理器设计的)。

站在 21 世纪的角度看来,权衡的结果就不一样了。对于大多数类型的 CPU,为了避免当程序员在需要注意高速缓存时因疏漏而导致的系统软件出错,在硬件复杂度方面付出一定的代价看起来更加划算。在本书写作的时候(2006 年),除了最小的 MIPS CPU 之外,有一种趋势在朝着用一定程度硬件管理实现“透明的”高速缓存的方向发展。如果你的 CPU 实现了完全一致的高速缓存,你可能根本不需要阅读本章(但是很少有 CPU 完全做到这一点)。在未来几年内面世的许多 MIPS 核仍然不会有一致的高速缓存控制器,所以这个过渡要花很长时间才能完成。

如果你的软件需要解决这些问题,就需要能够一个高速缓存项进行两种不同的操作。

第一个操作叫做回写操作。当感兴趣的数据位于高速缓存并且已经被污染(标记为自从上次从内存读入高速缓存或者从高速缓存写回到内存之后,又被 CPU 修改),那么 CPU 要把数据从高速缓存写进主存。

第二个是作废*(invalidate)*。当感兴趣的数据在高速缓存里边,CPU 把它标

记为无效这样随后的访问就会从内存中取出最新的数据。

在这个上下文中使用“flush”这个富有浓厚情感色彩的词更有诱惑力。但是这个词有容易引起歧义，它可以指回写、作废或者两个操作的结合——所以本书避免用它。

当有两个或者更多个处理器共享内存时就会涉及到一些复杂得多的问题。大多数共享内存的系统过于复杂以至于一个 CPU 没法知道别的 CPU 要读写的位置，所以软件没法确切知道哪个内存区域需要作废或者回写。要么共享内存访问永远不进行高速缓存（除非交互很有限否则很慢），要么高速缓存必须有特殊的硬件保持高速缓存（和内存的）一致性。多 CPU 的高速缓存一致性问题在 1980 年代中期由一群搞 FutureBus 标准的工程师进行系统化之后没有以前那么可怕了。在第 15.3 节对多处理器机制作了讨论。

4.7　二级（L2）和三级（L3）高速缓存

在大型的系统中，通常需要有嵌套的多级高速缓存。一个小而快的一级（L1）即主高速缓存离 CPU 最近。访问 L1 未命中时，不是直接从内存中查找而是先查找 L2 即二级高速缓存——通常比一级缓存要大几倍也慢几倍（但是比内存还是要快好几倍）。高速缓存需要几级取决于主内存速度和 CPU 最快访问速度比起来有多慢；由于 CPU 时钟周期时间比内存的时间周期缩小得快得多，2006 年的桌面系统普遍拥有三级高速缓存。嵌入式系统要落后于桌面系统几年（还有不同的约束，特别是功耗和散热方面），除了几个专门的高端应用外，2006 年才刚刚到达采用二级高速缓存的地步。

4.8　MIPS CPU 的高速缓存配置

我们现在可以把从古到今一些典型的有代表性的 MIPS CPU，按照其采用的高速缓存实现方式进行分类，来看看各级高速缓存的演化（参见表 4.1）。

表 4.1: MIPS CPU 高速缓存的演化

	L1				L2			L3		
	容量		几路	片上		几路	片上		几路	片上
CPU(MHz)	I-cache	D-cache	相联?	集成?	容量	相联?	集成?	容量	相联?	集成?
R3000-33	32K	32K	1	否						
R3052-33	8K	2K	1	是						
R4000-100	8K	2K	1	是	1M	1	否			
R10000-250	32K	32K	2	是	4M	2	否			
R5000-200	32K	32K	2	是	1M	1	否			
RM7000-250	16K	16K	4	是	256M	4	是	8M	1	否

随着时钟的速度的提高，我们看到越多样化的高速缓存配置，因为设计者设法应对速度比内存系统跑得越来越快的 CPU。为了维持运行的顺畅，高速缓存必须通过明显快于外面下一级存储器的速度提供数据，同时也要在多数情况能成功提供数据（即命中）来提高性能。

加上了二级高速缓存的 CPU 可以减少一级高速缓存未命中的代价——这样一级高速缓存就可以做得更小或更简单。典型的具有片上 L2 高速缓存的 CPU 都拥有更小的 L1 高速缓存，双 16-KB 的L1 高速缓存成为极受欢迎的“香饽饽”。直到最近为止，大多数 MIPS CPU 集成的 L1 高速缓存都可以在一个时钟周期内完成访问。有理由认为，随着芯片性能的增长以及时钟频率的提高，在一个时钟周期内能够存取的存储器大小基本上是个常数。但是事实上，片上存储器速度要滞后于逻辑。现代的 CPU 设计经常加长流水线主要是为了能够允许双周期的高速缓存访问。[4]

片外高速缓存常常是直接映射的，因为组相联的高速缓存需要宽得多的接口以传送多个标签并行匹配。

在所有这些演化中，主要有两代和 MIPS 高速缓存的软件接口。一种是由 R3000 开创并被大多数早期 32 位的 MIPS CPU 所追随的风格；还有一种是由 R4000 开始的，在到目前为止所有 64 位 CPU 中采用的风格。R4000 式的高速缓存现在也成了 MIPS32 标准的一部分，现在已经普遍使用了——我们只介绍这一种。

大多数现代的 MIPS CPU 拥有回写式、虚拟索引的、物理标签、两路或者四路组相联的一级高速缓存。高速缓存管理通过一条特殊的 **cache** 指令进行。

第一批 R4000 CPU 拥有片上二级高速缓存控制器，QED 的 RM7000（1998 年前后）引入了片上二级高速缓存，这在当今的高端设计中很常见。MIPS32/64 定义的 **cache** 指令也适用于扩展到 L2 和 L3 高速缓存。

注意!　有些系统的二级高速缓存不是由 MIPS CPU 内部的硬件来控制的，而是建立在存储器总线上。对于这类高速缓存的软件接口将与具体系统有关，和本章介绍的由 CPU 实现的或者 CPU 控制的高速缓存相比，可能有很大的不同。

4.9　对 MIPS32/64 高速缓存的编程

MIPS32/64 定义了一个非常简洁的方法来管理高速缓存（在 R4000 之后，他们对早期 CPU 高速缓存中那种不像样的维护方法作了修正）。

MIPS32/64 CPU 常常会有比以前的 CPU 复杂得多的高速缓存——采用回写方式并且高速缓存行更长。因为是回写式的高速缓存，每个高速缓存行就需

[4]这不是最近才发明的。采用 RISC HP-8x00 CPU 系列制造的早期系统接受一个双周期 L1 高速缓存延迟已换取容量巨大的外部一级高速缓存，而且性能不错。

要一个状态位，以便在该行被 CPU 写入时（此时其内容和相应主存的数据不同了）标记被写过。要管理高速缓存，有几个基本操作是非要不可的。我们要能做到以下几点：

- 作废某块高速缓存区域：将该地址范围内的数据清除出高速缓存，这样下一次引用的时候就会从内存获取最新的数据。

  指令 **cache HitInvalidate** 形式和 load/store 指令一样，提供一个虚拟地址。该指令作废包含虚拟地址所引用数据的高速缓存行。在整个地址范围内每隔一个缓存行大小的地址上重复该指令。

- 回写一段地址区域：保证 CPU 向该地址范围写入的数据如果保存在已经修改的缓存行中就被送回到主存。

  指令 **cache HitWritebackInvalidate** 回写包含虚拟地址所引用相应数据的高速缓存行，顺便把它标记为作废。

- 作废整个高速缓存：丢弃高速缓存的所有数据。除了初始化之外极少需要这个操作，但是有时操作系统搞不清到底要作废哪部分高速缓存时，也用它作为最后一招。

  指令 **cache IndexInvalidate** 就是在这种情况下使用的。其地址参数仅仅解释为对构成高速缓存的存储芯片的地址索引——后续的高速缓存行从索引值 0 向上计数，每次以高速缓存行的大小递增。

- 初始化高速缓存：从一种未知状态把高速缓存带入可用状态所必需的全部操作。设置高速缓存控制域（即“标签”）通常涉及到对 CP0 **TagLo** 寄存器清零，以及对每一高速缓存行上执行 **cache IndexStoreTag** 指令操作。带有数据校验位的高速缓存还需要用数据填满——尽管高速缓存都是无效的，填充的数据也不应当有错误的校验位。填充 I-cache 需要特别小心，但是通常可以用一条 **cache Fill** 指令。

4.9.1 cache 指令

cache 指令的形式和 MIPS 的 load/store 指令的通用形式相同（采用通常的寄存器加上十六位有符号数的偏移量进行寻址）——但是对应于数据寄存器编码的地方是个 5 位的操作域，用以编码待操作的高速缓存、确定怎样找到相应高速缓存行、以及找到相应行之后怎么处理。用汇编写一个高速缓存行就像 **cache OP, addr** 这样，其中 OP 只是一个代表操作域的数值。

最好的做法就是使用一个 C 语言的预处理程序定义一个文本的“名字”代表相应操作的数值。对此没有标准的命名；我就随便用了在 MIPS 开工具包的头文件里找到的 C 预处理中定义的名字。

该 5 位域的高 2 位选择要操作的是哪个高速缓存：

0 = L1 I-cache

1 = L1 D-cache

2 = L3 如果有的话

3 = L2 如果有的话

在我们列出这些操作之前，先说明一下，每个操作都有三种形式，区别在于怎样选择要操作的高速缓存项（即高速缓存行）的方式：

- 命中型操作：给出一个地址从高速缓存中查找。如果找到（即“命中”）则执行操作，否则什么也不干。
- 寻址型操作：给出内存中某个数据的地址，处理就和经过高速缓存访问一样。就是说，如果你寻址的行不在高速缓存，在执行高速缓存操作之前要从内存取出数据。
- 索引型操作：根据需要用虚拟地址的低位依次选择高速缓存行内的字节，然后是高速缓存某一路内的行地址，然后是该路。[5] 你必须知道高速缓存的大小（可从 **Config1-2** 得知，详情参阅第 3.2 节）才能知道各个域的确切边界，但地址用法跟下面差不多：

31 … 5			4 … 0
Unused	Way1–0	Index	byte-within-line

一旦选定了相应的高速缓存以及某一行，你有几个操作可以选择，如表 4.2 所示。CPU 必须支持这三种操作才算得上是 MIPS32/64 兼容的：索引型作废、索引型存储标签、命中型回写作废。其余操作都是可选的——如果要用的话，清仔细对照 CPU 手册。

synci 指令（MIPS32 第二版新增的）为保证你刚刚写入的指令能够正确执行（合并了一个 D-cache 回写和 I-cache 作废操作）提供了一个明确清晰的机制。如果你的 CPU 支持，就应当使用 **synci** 而不是传统的替代方案（用一对 **cache** 指令来做：D-cache 回写后跟一个 I-cache 作废）。

4.9.2 高速缓存初始化和 Tag/Data 寄存器

能够读写高速缓存标签的能力有助于诊断和维护。MIPS32/64 定义了一对 32 位寄存器 **TagLo** 和 **TagHi**[6] 在高速缓存的标签部分和管理软件之间倒腾数

[5] 有些还在用的 MIPS CPU 使用地址的最低有效位确定选择哪路高速缓存。这些 CPU 可能需要特殊的初始化。我不知道有哪个符合 MIPS32/64 的 CPU 是这样做的：但这是另一回要小心的事了。

[6] 有些 CPU 手册采用不同的寄存器和 I-cache 和 D-cache 通信，分别叫做 **ITagLo** 等等。

表 4.2: 每个高速缓存行上可用的操作

取值	命令	功能
0	Index invalidate	设置该高速缓存行为“无效”。如果是 D-cache 的行有效并且被修改过污染（从内存取出后已被写入），那就先把其内容写回内存。 这是初始化 CPU 时作废指令缓存最佳也是最简单的方法——当然如果高速缓存有奇偶校验保护，你还要填入带正确校验位的数据。参见下面的 Fill。这对数据高速缓存 D-cache 不适用，因为可能导致随机的回写：参加下面的 IndexStoreTag 类型。所有 MIP32/64 的 CPU 都必须提供这一操作。
1	Index Load Tag	读取高速缓存行的标签位及其所寻址的双字数据到 TagLo/TagHi 寄存器。这个命令比较生僻，仅供诊断和特殊癖好者使用。
2	Index Store Tag	用 TagLo/TagHi 寄存器的值设置高速缓存标签。要从未知状态初始化数据高速缓存，把 TagLo/TagHi 寄存器设为零，然后逐行执行该操作。所有 MIP32/64 的 CPU 都必须提供这样的操作。
4	Hit invalidate	作废但并不回写数据（即使已经被污染）。所有 MIP32/64 的 CPU 都在指令缓存上实现这个操作。除非知道数据没有被污染（修改），否则有可能导致数据丢失。
5	Hit Writeback invalidate	作废该行——但是如果被污染要先回写数据后再作废。在运行中的数据高速缓存这是推荐的作废方法。所有 MIP32/64 的 CPU 都在 D-cache 实现本操作。
5	Fill	地址类型操作——用提供的地址处的数据填入合适的高速缓存行——就象在该地址处理 I-cache 未命中那样选择。 用来初始化指令高速缓存行的数据域，应该在奇偶校验保护下设置 CPU 时完成。
6	Hit Writeback	如果该行被污染，回写到内存但是让高速缓存保持有效。在运行的系统中你想保证数据确实已经被推入内存以便被 DMA 设备或者其它 CPU 访问时使用该指令。
7	Fetch and Lock	地址型操作。把该地址处的数据加载进象正常高速缓存引用时同一高速缓存行（如果数据还不在高速缓存，可能要把先前的数据回写）。 然后锁住该行。被锁住的行在高速缓存未命中的时候不被替换。 一直保持锁定状态，直到使用某种 **cache**“invalidate”明确作废为止。

据。**TagHi** 常常不是必要的：如果你的物理地址空间不超过 36 位，**TagLo** 通常就能为整个标签提供足够的空间。

这两个寄存器中的域反映了高速缓存中的标签域，因而依赖于具体 CPU。唯一：可以保证的一点就是：Tag 寄存器中全零的值对应于一个合法有效的数据标签，表示相应的高速缓存行不含有效数据。CPU 实现一条 **cache IndexStoreTag** 指令，该指令把标签寄存器的内容传送到高速缓存行。所以通过将寄存器的值设为零，并且存入标签值，你就可以从一种未知状态开始初始化高速缓存。

MIPS32/64 定义了“存储数据”和“装载数据”的高速缓存操作，但都是可选的，只能由针对具体 CPU 的代码使用。你永远可以通过“命中”数据来访问数据。

一个高速缓存标签必须保持所有不能从高速缓存索引中直接得到的地址位（如果觉得不清楚请参见图 4.1）。

所以如果一级高速缓存标签域长度就是所支持的物理地址的位数和用来索引一级缓存的位数之差——对 16K 的四路组相联高速缓存来说就是 12 位。**TagLo(PTagLo)** 可以容纳 24 位，这样一个高速缓存最多可以支持 36 位物理地址范围。

TagLo(PState) 域包含有状态位。你的 CPU 手册会告诉你更多细节；然而对于所有高速缓存管理和初始化来说，知道全零的标签永远一个代表相应高速缓存行内容作废的有效标签就够了。

4.9.3 CacheERR、ERR 和 ErrorEPC 寄存器：存储器/高速缓存出错处理

CPU 的高速缓存是存储系统至关重要的一部分，对于高可用度或高可信度的系统会发现用额外的位来监控存储在那里的数据的完整性是值得的。

理想的数据完整性检查的实现应当是端对端的；数据一产生或一进入系统就计算校验位并随数据一起存放，在数据使用前再执行检查。这样检查将不仅会发现存储器的错误而且能发现复杂的总线错误，以及数据一路到达 CPU 和返回途中所出现的其它错误。因为这个原因，可用于实现高可靠系统的 MIPS CPU 常常选择在高速缓存里提供错误校验。和主存系统一样，你既可使用简单的奇偶校验也可以用复杂的纠错码（ECC ——Error Correction Code）。

如今的日子里，构建存储系统的元件倾向于以多个 8 位宽度的形式提供，允许校验的内存系统模块提供 64 位数据和 8 位校验位。所以我们用的别的东西也要遵循这一点。

奇偶校验简单地为内存中每个字节实现一个额外的位。一个奇偶错误告诉系统这个数据不可靠，允许在某种程度控制下关闭系统而不是潜入随机错误。奇偶校验的一个重要的作用就是在系统开发的过程中提供巨大的帮助，因为它能明确断定问题是出在内存数据的完整性上。

完全是垃圾数据的一个字节有百分之五十的概率产生一个正确的奇偶校验位，在 64 位总线上的随机垃圾数据及其校验位每 256 次中将有 1 次逃脱检测。有些系统要求比这更高。

纠错码计算起来更为复杂，因为涉及到整个 64 位的字和 8 个校验位。这样更为彻底：1 位的错误将被唯一的识别并纠正，任何 2 位的错误也都能检测出来。在非常大的存储器中，ECC 是排除随机错误的强大工具。

因为纠错码一次可以检查整个 64 位的数据字，所以使用纠错码 ECC 的内存不能采用只选择一个数据字的一部分的方式进行部分字数据的写操作，而只能采用合并新的数据并重新计算纠错码的方式。不用高速缓存时运行的 MIPS CPU 要求存储器系统能进行部分字数据的写操作，这将使事情变得复杂化。存储器系统硬件必须将部分字数据的写操作转变为“读取-合并-重新计算-写入” 的操作序列。

对于简单的系统通常选择奇偶校验位或者干脆什么都没有。让校验作为可选项是很有意义的，这样在设计研发过程中有利于诊断，而在成品时却不用付出相应的代价。

理想情况下，存储器系统次采用的什么校验机制，高速缓存系统也就采用什么。根据不同的 CPU，你可以采用奇偶位、每个双字 8 位纠错码、或者干脆什么保护都不用。

如果可能，数据校验位通常直接通过系统接口送到高速缓存：数据从存储器到达加载进高速缓存时不需要检查。数据在使用的时候才进行检查，这样可以保证任何奇偶异常都被转交给引起异常的指令，而不是转交给碰巧位于同一缓存行的指令。

系统总线上的数据校验错对于非高速缓存的数据存取采用同样的处理机制。这意味着会被报告成“高速缓存奇偶错”——容易让人搞糊涂。

注意，系统接口把进来的数据标记为没有有效的校验位是可能的。在这种情况下，CPU 会为它内部的高速缓存生成校验位。

如果发生错误，CPU 会生成一个特殊的错误自陷。这时控制被指引到非高速缓存空间专门的位置（如果高速缓存含有错误，继续从里头执行代码显然很愚蠢）。如果系统采用 ECC，硬件要么自己纠正错误，要么提供给软件足够的信息去修正错误。

CacheErr 寄存器的域取决于具体实现，你需要查阅 CPU 手册。你也许能够从你的 CPU 供应商那里得到高速缓存错误管理的例程的示例代码。

4.9.4 确定高速缓存的大小和配置

对于符合 MIPS32/64 兼容的 CPU，高速缓存的大小、组织、以及行宽都可以作为 3.3.7 节所讲的 CP0 **Config1-2** 寄存器的一部分可靠给出。

但是为了兼容性，书写或者复用能够在很广泛的范围的 MIPS CPU 上都能可靠运行的初始化软件更有意义。下一节讲些尝试和测试的解决方法。

4.9.5 初始化程序

这儿将介绍一个很好的做法：

1. 开辟一些内存以便可以从它来填充高速缓存——你存入什么数据没有关系。但如果你的系统使用奇偶校验码或是纠错码，你必须确保校验位是正确的。一个好的窍门是至少到高速缓存初始化完成之前预留系统内存的低 32K 用于此目的。如果用不经高速缓存写入的数据来填充，就会包含正确的校验位。

   对于初始化大的二级高速缓存，那样大小的缓冲区不够，你可能需要做更为复杂的事情。

2. 将 **TagLo** 寄存器设为零，这样能保证对应行有效的那一位不会置位并且标签的奇偶码是一致的（在需要的 CPU 上也要设置 **TagHi**）。

   TagLo 寄存器将被 **cache IndexStoreTag** 指令用来强制作废对应的高速缓存行并清除标签的奇偶校验位。

3. 禁止中断以防万一发生。

4. 先初始化 I-cache，然后是 D-cache。下面是初始化 I-cache 的 C 代码。（你必须相信象 `Index_Store_Tag_I()` 这样的执行底层功能的函数或宏；它们要么是运行在相应机器指令的琐碎的汇编代码例程，要么是——对于大胆的 GUN C 用户——调用 C 嵌入汇编 asm 语句的宏。）

```
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    /* clear tag to invalidate */
    Index_Store_Tag_I(addr);
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    /* fill once, so data field parity is correct */
    Fill _I (addr);
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    /* invalidate again    prudent but not strictly necessay */
    Index_Store_Tag_I();
```

   我们执行填充操作，是因为有些 CPU 甚至对明显无效的高速缓存行也可能检测奇偶错误并发生自陷。不幸的是，MIPS32/64 没有强制要求实现 **Fill_I** 操作。你可以合理的期望任何实现了奇偶校验和 ECC 保护的 CPU 都应该包含了该操作：不保护高速缓存数据的 CPU 只需要第一个存储标签的循环。

   还有，我们用了三个分别的循环，而没有合并他们，因为你在处理标签时必须非常小心；对于两路相联的高速缓存，单个循环会对一半的高速缓存

初始化两次，因为用零标签寄存器的“index store tag”会复位 LRU 位，而该位决定哪组高速缓存行在下一次未命中时使用。

5. D-cache 的初始化稍微要复杂一些，因为没有对应数据方的 fill 操作；我们只能通过从高速缓存加载而依靠通常的高速缓存失效处理。下面是具体的方法。

```
/* clear all tags */
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    Index_Store_Tag_D (addr);
/* load from each line (in cached space) */
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    junk = *addr;
/* clear all tags */
for (addr = KSEG0; addr < KSEG0 + size; addr += lnsize)
    Index_Store_Tag_D (addr);
```

4.9.6 作废或者回写高速缓存中的存储区

用于作废或回写的参数一律对应于一些 I/O 缓冲区的程序地址或物理地址的范围。

几乎总是用命中型的操作来作废或回写。如果你需要作废或回写一大块存储区，使用索引操作来作废或回写整个高速缓存会比较快，但这只是提供一个优化的方法，你完全可以选择忽略本方法。

下面这样做就够了：

```
PI_cache_invalidate(void *buf, int nbytes)
{
    char *s;

    for (s = (char*)buf; s < buf + nbytes; s += lnsize)
        Hit_Invalidate_I (s);
}
```

只要 buf 是程序地址就没有必要生成特殊的地址。如果你不得不基于物理地址 p 来作废，只要 p 在物理空间的第一个 512M 之内，加上一个常量就能生成相应的 kseg0 区地址：

```
PI_cache_invalidate (p + 0x80000000, nbytes);
```

4.10 高速缓存的效率

从九十年代早期转到片上高速缓存，高端 CPU 的性能很大程度上取决于其高速缓存系统的性能。现在许多的系统（尤其是嵌入式系统，需要在高速缓存容量和内存性能上节约），CPU 有 50－65% 的时间时在等待高速缓存重填。就这点来说，将 CPU 核性能翻倍只能将应用程序的性能提高 15－25%。

高速缓存的效率取决于系统在等待高速缓存填充花费的时间。你可以将之归结为两个数的乘积：

- 平均每条指令的高速缓存未命中率：高速缓存未命中的次数除以执行的指令数。
- 高速缓存未命中/重填的开销：存储器系统重填高速缓存并再次开始 CPU 执行所花的时间。

你可能会想，考虑高速缓存未命中率——每次内存访问的平均未命中率。但是高速缓存未命中可能有许多因素影响，其中有些根本没法预料。举个例子，x86 CPU 的寄存器个数很少，所以同样的程序编译给 x86 使用会比给 MIPS 使用多很多的数据存取。但是 x86 使用堆栈来代替寄存器给这些额外存取使用；而这堆栈位置将是内存中使用非常频繁的区域，对应的使用效率会非常高。每千条指令的未命中率受这种差别的影响就没那么大。

即使象上面这样简单的分析对于下面指出的几种加快系统速度的显而易见的方法很有帮助。

- 减少高速缓存未命中的次数：
  - 加大高速缓存。这总是有效的，但是价格昂贵。64K 大小的高速缓存占据的芯片面积大约等于甚至超过简单的 CPU 中其它所有部分（不含浮点硬件）占据面积加起来的总和。
  - 增加高速缓存的相联度。增加到四路相联还是值得，再增加下去性能提高就几乎看不到了。[7]
  - 再加一级高速缓存。当然这将使计算变得更加复杂。除增加了另一个子系统的复杂度外，二级高速缓存的命中率低得让人沮丧；一级高速缓存已经从 CPU 的重复数据访问行为中“捞走了油水（原文为 skimmed the cream，直译为撇去了奶油）”[8] 。为了使其物有所值，二级高速缓存必须比一级大许多（一般来说是八倍或者更大），并且二级缓存命中的访问时间必须比内存快（两倍或者更多）。

[7]确实有八路或者更多路相联的高速缓存，但是通常这样做是因为别的原因而非减少高速缓存未命中率。提供充足的高速缓存相联路数可以有效降低功耗（当不用的时候整路都可以关掉），有时候也可以避免第 4.12 节提到的高速缓存重影问题。

[8]感谢 Hennessy 和 Patterson 提出的这个形象的比喻。

- 重新组织你的软件从而减少缓存未命中的次数。不清楚这在实践中是否有效：对于小的程序很容易重新组织取得显著效果，但迄今为止还没有人成功做出过对任意程序都有效的通用工具。参见第 4.11 节。

- 减少高速缓存重填的开销：

  - 更快的让 CPU 获得第一个字的数据。DRAM 存储器系统必须做许多工作来启动，然后才能快速提供数据。内存距 CPU 越近，它们之间的路径越短，数据到达就越快。

    增加带宽需要花钱。而另一方面，减少时延通常可以通过简化系统来达到：所以这是此处列出的唯一既可以省钱又可以同时提高性能的方法。不过奇怪的是这点很少被注意到，也许是因为它需要在 CPU 接口和存储器系统设计之间考虑更多的集成。CPU 设计者在确定芯片的接口时不愿处理系统问题，可能是因为他们的工作已经够复杂的了!

  - 增加内存突发的带宽。这传统上是通过昂贵的“交替存储”技术做到的：用两个或者更多的存储器来交替存储数据字；经过初始的延迟后，就可以交替着从每一个存储器中取数据，让可用带宽翻倍。存储器时延随着芯片缩小的降低极为缓慢，然而从单个的标准存储器部件可得到的带宽却在爆炸性增长。而且在一个 DRAM 内存条里，芯片对外呈现出可以做流水线化访问的分开的内部存储块以降低总体时延。因而，物理的交替存储现在很少见了。

- 早点重新开始运行 CPU：天真的 CPU 在整个高速缓存行填满新数据之前就会傻等。但是你可以安排让来自内存的数据在传递到高速缓存的同时直接送到等待的 CPU，等待的数据一到就让 CPU 立即开始执行。高速缓存其余部分的重填和 CPU 活动可以并行进行。许多 MIPS CPU 对这一点做了增强，传递触发高速缓存未命中请求的字的地址到内存控制器并让块中的紧要的字首先返回（“紧要字优先”）。

- 直到 CPU 必须使用数据的时候才停下：更为激进的方式是让 CPU 绕过取数操作继续执行下去；取数的操作交给总线单元，CPU 继续运行直到它真的需要引用被读取的数据。这被称之为无阻塞读取，现在已经是普遍做法。

  在正常的 CPU 上，以及大多数的软件中，实际上走不了多远，就会碰到有指令要用到刚刚加载的数据而使得 CPU 反正得停下来。但这种做法有效地减少了几个周期的时延，在有二级高速缓存的时候可能特别有用。

  最为激进的方法是，可以执行任何后续的不是在等待被加载数据的代码，R10000 的乱序（OOO）执行就是这样做的。OOO 设计普遍采用这个技术，不光用于读取数据还用于计算和分支指令。现在最快的高端 CPU

都实现了 OOO 流水线。不错，OOO 不仅复杂而且功耗大，这点可能会延缓它进入功耗敏感的嵌入式应用，但仅仅是个时间问题。

- 多线程化 CPU：高速缓存未命中引起的时延对程序性能的不利影响只能缓解却不可避免。所以真正激进的方法是在 CPU 上运行多个线程，可以在互相等待的时间里利用空闲的 CPU 资源。参阅附录 A 对 MIPS MT 系统的简介。

4.11 重新组织软件以影响高速缓存的效率

在可能运行大量事先无法确定的应用程序的系统中，只能对高速缓存的行为进行估计。但在运行单个应用程序的嵌入式系统中，高速缓存未命中的模式取决于某个特定软件的具体构建过程。这一点特别让人容易联想到是否能够用一种系统的方式对应用程序代码做一下按摩以改进高速缓存的效率。要了解怎样才能有效做到这点，需要按照产生的原因对高速缓存未命中分类：

- 第一次访问：任何数据必须至少从内存中读入一次。
- 替换：高速缓存的大小是有限的，所以你的程序没有运行多久就会出现，高速缓存未命中和重新填充，会把一些有效的数据替换出去，其中一些数据是值得保留在高速缓存中的。当程序运行过程中，就会不停的损失数据然后又不得不重新加载。你可以通过使用加大高速缓存和减小程序的大小（最终起作用就是程序大小和高速缓存大小之比）来使替换导致的未命中减到最少。
- 抖动。在四路组相联的高速缓存中（更多路的情况在 MIPS CPU 中很少见），所以对于任何内存地址在高速缓存中最多有四个位置可以存放。（在直接映射的高速缓存中就只有一个）。

  如果你的程序碰巧会非常频繁的使用某几块数据，而这几块数据的地址低位又非常接近，以至于它们恰好使用相同的高速缓存行，那么当数据块的数目超过高速缓存组相联的路数时，你就会在某些时间段内碰到极为频繁的高速缓存未命中，因为不同块的数据会互相把对方不停地排挤出高速缓存。

  抖动损失随着组相联的数目急剧减少；大多数的研究表明超过四路的高速缓存对于性能几乎没有提高（但是可能有其它的别的原因需要构建八路甚至更多路的高速缓存）。

有了上面的背景知识，那么针对程序做怎样的变化使它在高速缓存中运行得更好？

- 使程序更小：如果能做到的话，这是个好主意。你可以使用编译器的中度优化选项（过度的优化常常使程序更大）。

- 让程序中经常执行的那部分更小：程序中的访问密度完全不是平均分布的。经常会有相当一部分代码几乎从不用到（错误处理，生僻的系统管理），或者只用一次（初始化代码）。如果你能剥离出来这些很少用到的代码，剩余部分的代码就能有很高的高速缓存命中率。

  一种经过实验证明在一定程度上取得成功的方法是，利用剖析器确定程序在典型的负载下使用最频繁的函数，然后按照函数的执行时间递减的顺序在内存中安排函数。这样至少这些经常使用的程序不会相互争夺高速缓存的位置。

- 强迫一些重要的代码或数据常驻高速缓存：有些厂商提供了一种机制能允许一部分高速缓存加载数据后不被替换。这看上去好像是为中断处理程序或是其它重要软件获得确定性能的一种方法。通常的实现是占用多路组相联的高速缓存其中一路。

  我很怀疑这个方法的有效性，我也不知道有任何研究结果支持它的有效性。系统其余部分的性能损失很可能超过了那些关键代码的性能改进。高速缓存锁定被用作一个非常可疑的营销手段来打消客户对高速缓存启发式特性的顾虑。这个顾虑是可以理解的，但是更快速、复杂、大型的系统似乎本质上就是不可预测的，大多数开发人员也许应该学会接受而不是避免它——高速缓存毕竟只是问题的一部分。

- 安排程序的布局避免抖动：除了让程序的活动部分变得更小（参见上文）之外，我觉得这会导致太多维护问题，所以不是个好办法。（即使是两路）组相联的高速缓存让这种做法失去了意义。

- 让那些很少用到的数据和代码不经过高速缓存：把高速缓存只保留给那些重要的代码或数据使用，排除那些只用一次或很少使用的代码。

  但这几乎总是一个错误。如果数据真的很少使用，那么本来就不大会进入高速缓存。因为高速缓存通常总是以长度 4–16 字的行为单位读取数据，所以即使是遍历只用一次的数据或代码也能有巨大的加速；从内存突发填充比单个字的存取访问几乎不多花时间，还能给你顺便提供另外的 3–15 个字的数据。

简而言之，我们热情推荐以下面的方法作为一个起点（经过多次实践测量和深入思考后才才可以放弃）。首先，让除了 I/O 寄存器和用得不多的远程存储器之外的一切访问都使用高速缓存。观察一下看看高速缓存对你的应用程序有什么影响，而不要作事后猜测。其次，用硬件解决硬件问题。没有任何软件能够挽回因为高速缓存重填时延过长或者内存带宽过低而损失的性能。试图重新组织

软件而增加高速缓存命中率，必然耗时而且复杂。一开始就要明白这样做的收获甚小而来之不易。也试试硬件办法!

4.12 高速缓存重影

这个问题只会影响这样一类的高速缓存，用于生成高速缓存索引的地址和存放在高速缓存标签内的地址不一样。采用虚拟地址生成索引而用物理地址作为标签，是 MIPS CPU 一级高速缓存的常见做法。这对性能有好处：要是用物理地址生成高速缓存索引，我们要一直等到地址经 TLB 转换之后才可以开始查找高速缓存。但这样可能导致高速缓存重影*(aliases)*问题，如图 4.3 所示。

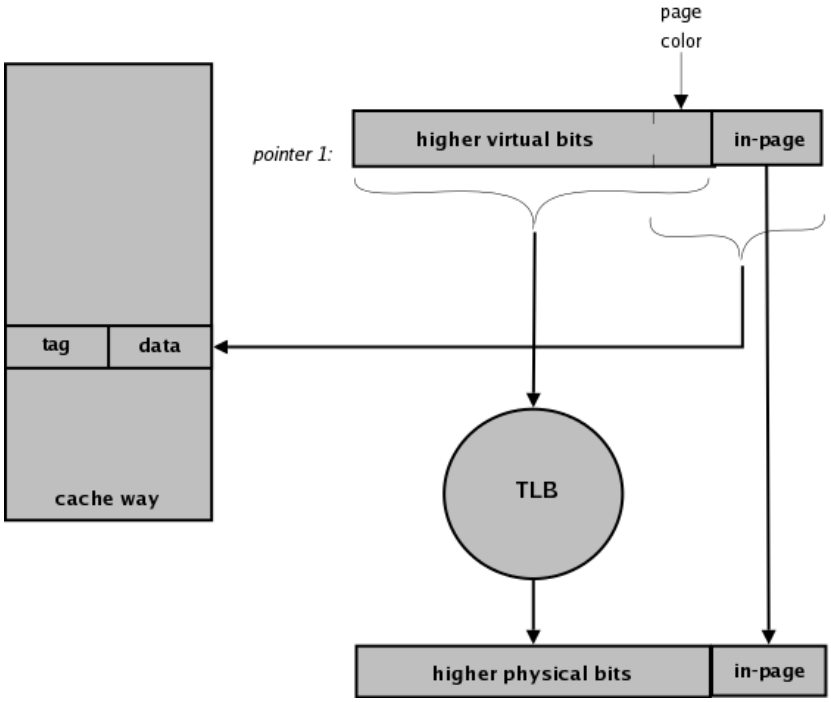

图 4.3: 高速缓存重影

大多数这些 CPU 按照 4KB 的页大小来转换地址，这就是说，虚拟地址的低 12 位不需要转换。只要你的高速缓存是 4K 或者更小，虚拟的索引和物理的索引就是一致的，那就没有问题。其实这个条件还可以放宽：只要高速缓存索引的跨度范围是 4K 或者更小，就没有问题。在组相联的高速缓存中，每个索引访问几个高速缓存槽（每路一个）——所以即使是 16K 的四路组相联高速缓存，虚拟索引和物理索引仍然相同，不会发生重影。

但是设想一下你的高速缓存索引范围达到 8K 的情形（32K 四路组相联的高速缓存就是这样）。你可能从两个不同的虚拟地址访问同一个物理页：例如虚拟

地址可能是连着的两页——比如说地址分别从 0 和 4K 处开始。如果程序访问 0 处的数据，该数据就加载到某个高速缓存槽的索引为 0 的位置。如果此时再从另外一个地址 4K 访问同一个数据，再次从内存中取出放入高速缓存的不同索引 4K 处。现在同一个数据在高速缓存中有两个影像，对其中一处的修改另一处无法知道。这就是高速缓存重影。只读数据的重影容易引起混乱也有可能导致错误，而对于正在写的数据的重影就是一个随时可能起爆的炸弹。

MIPS 的 L2 级高速缓存总是物理索引和物理标签，所以不会有重影的问题。[9]

缓存重影问题是偶然因素引入到 MIPS 世界中的。只要你使用了外部的二级高速缓存，原始的 R4000 CPU ——上面提到的——是没有重影的。但是后来发现 R4000 可以用来构造没有二级高速缓存的小型廉价系统。这就暴露出了重影问题；但是 64 位 MIPS CPU 想要征服世界的愿望压倒了谨慎从事的作风，重影问题被重新定义为由系统软件来克服的问题。

重影不会出现在一对生成相同高速缓存索引的虚拟地址之间。对于 4-KB 的页面，高速缓存索引的低 12 位保证是相等的；唯一必要的是保证对应于同一个物理页面的不同程序地址间的距离为 L1 高速缓存每路最大容量的整数倍。如果你的操作系统在设置多个虚拟地址到同一个物理页的映射时小心地保证虚拟页地址的间距为 64K 的整数倍，就很难想象你还会出问题。[10]

所以只要 OS 对于同样的物理数据给出多个虚拟地址时充分小心，避免重影是可能的。不幸的是，Linux 是一个可移植的操作系统，那帮开发人员考虑更多的是通用的桌面或者服务器系统的问题，而不知道（或者不关心）只影响 MIPS CPU 的特定问题。这样 Linux 内核避免了大多数重影，但是试图保留其余的重影。

简单的操作系统要么没有多重映射的页，要么能够在这种约束下工作。但是高速缓存重影是操作系统很多潜伏的问题和错误的一个来源，要是硬件设计人员当年记得这其实是个缺陷该多好啊！只是因为十五年前的一个阴差阳错，一直误把缺陷当成了特色...

[9] 带有片上二级缓存控制器的 CPU 可以用二级缓存中的一些位跟踪进入一级高速缓存的读取；R4000 和 R4400 CPU 用这个来检测高速缓存重影并产生特殊的异常让系统软件解决这个问题。但是这个传统似乎没有被后来的 CPU 传承下来。

[10] 虽然 CPU 的集成度和速度每过一年都会无情地增长，但一级高速缓存每一路的容量不大可能超过目前在 64K 四路组相联高速缓存中看到的 16 KB 记录。L1 高速缓存以 CPU 时钟速度全速运行，越小就越快；以后集成化程度更高的 CPU 将会转而追求片上的 L2 高速缓存。

第 5 章　异常、中断及初始化

在 MIPS 体系结构中，中断、自陷、系统调用以及其它打断程序正常执行流的事件统称为异常，都采用同一种机制处理。都是些什么样的事件呢？

- 外部事件：在 CPU 核之外的事件——即来自于真实的“连线”上的输入信号。这些就是中断。[1] 中断用来让 CPU 的注意转到某些外部事件：这是要同时处理多个事件的操作系统所必不可少的特性。

  中断是唯一由 CPU 正常指令流以外的事件引起的异常条件。因为中断不是我们小心就能避免的，所以必须有某种软件机制在必要时禁止中断。

- 存储器地址转换异常：当某个地址需要转换但是硬件不能有效转换时，或当写一个有写保护的页时，会发生这个异常。

  OS 必须确定这个异常到底是不是错误。如果异常症状是由于应用程序访问了允许的地址空间之外非法内存，问题的解决是操作系统终止应用程序以保护系统的其它部分。更常见的是良性的内存转换异常可以用来调用操作系统功能，复杂到可以是按需调页的虚拟存储系统，或者简单到只是扩充堆栈空间。

- 其它需要内核干预的非常情况：其中最为显著的一个例子就是浮点指令导致的条件，此时硬件无法处理某些困难和少见的操作符和操作数的组合而寻求软件仿真服务。

  这类情况比较模糊，因为不同的内核对这类情况会有不同的处理。未对齐的加载在一个系统中可能当作错误，在另一个系统中由软件处理。

- 程序或硬件检测到的错误：这些包括不存在的指令、在用户权限下非法的指令、在相应 SR 位被禁止时执行的协处理器指令、整数溢出、地址对齐出错、用户态中访问 kuseg 以外的地址。

- 数据完整性问题：许多 MIPS CPU 对来自总线和缓存的数据作字节奇偶校验或字宽的纠错码检验。高速缓存或者奇偶校验错在支持数据校验的 CPU 上产生一个异常。

[1] 还有些比较生僻隐晦的非中断的外部事件比如读取时报告的总线错——现在暂且把他们都当作一种特殊类型的中断。

- 系统调用和自陷：这些指令的整个目的就是产生定义好的异常；它们用来以一种安全的方式提供软件服务（系统调用、精心植入的条件自陷代码以及断点）。

有些事情不会产生异常，尽管与你的期望恰好相反。比如要检测写周期的总线错你就不得不用别的机制。那是因为大多数现代的系统对写操作进行排队缓冲：任何与写操作相关的的错误发生时间太晚以至很难判断出是哪条指令引发了错误。实在必须对写操作的错误进行通知的系统必须要用 CPU 外部的硬件，也许采用中断信号。

在这一章里，我们要看看 MIPS CPU 怎样决定接受异常以及软件要怎样做来正确处理异常。我们会解释为什么 MIPS 的异常叫做是“精确”的、讨论异常入口点并讨论一些软件的约定。

来自 CPU 外部的硬件中断是嵌入式系统中最常见的异常，对时间要求最严的、最有可能导致不易觉查的错误。*嵌套异常* ——在一个异常的处理还没完成之前又发生另一个异常，可能引起一些特殊的问题。

系统复位后 MIPS CPU 重新启动的方式是当作一种异常来实现的，借用了异常处理功能——所以也放在这一章讨论。

在这一章末尾，我们会看两个相关的话题：怎样用软件模拟一条指令（在指令集扩展机制中要用到），还有怎样构造一个信号量在面对中断时提供任务间的健壮的通信。第 14 章详细描述了在一个真实的、成熟的 MIPS 操作系统中是怎样处理中断的。

5.1 精确异常

在 MIPS 文档中会看到这样一个短语*精确异常(precise exception)*。这是一个有用的特性，但要理解为什么有用，你需要先看看替代方案。

在通过流水线（或者用更加复杂的指令重叠执行技术等）而获得最佳性能的 CPU 中，体系结构的顺序执行模型其实是硬件巧妙维护的假象。如果硬件设计不够巧妙，异常就可能导致该假象暴露/露馅。

当异常暂停正在执行的线程时，CPU 的流水线中还有几条处于不同阶段尚未完成/结束的指令。既然我们想要能够从异常返回并且不受破坏接着执行被打断的执行流，流水线中的每条指令要么执行完毕，要么就象我们根本没见过一样。[2] 此外，我们需要记住哪条指令属于哪一类型。

如果一个 CPU 体系结构给这个问题开出的解决方案让软件的生活尽可能的容易，那么它就具备精确异常的特性。在一个精确异常的 CPU 上，任何异常发生时，我们都指向一条指令（*异常受害指令*）。在该指令之前的所有指令都执

[2] 你其实可以搞得很复杂，让 CPU 保存一些中间结果，让异常处理程序解开乱麻又重新缠绕在一起，但谁会需要这么复杂呢？

行完毕，但是受害指令及其后续指令就好象从来就没有开始一样。[3] 当异常为精确的时候，处理异常的软件就可以忽略 CPU 实现的时序影响。

MIPS 体系结构接近于规定所有异常都为精确。全部内容如下：

- 明确的罪证：在任何异常之后，CPU 的控制寄存器 **EPC** 都指向一个正确的地方，异常处理之后从该处开始重新执行。在大多数情形，它指向异常受害指令；但是如果受害指令处于分支延迟槽之内，则 **EPC** 指向前面的分支指令：返回到分支指令会重新执行受害指令，但是返回到受害指令将导致分支被忽略。当受害指令处于分支延迟槽时，原因寄存器的 **Cause(BD)** 位置位。

- 异常出现在指令序列中：要是一个非流水线的 CPU 这点显而易见，但流水线中异常可能发生在几个不同的执行阶段，这会导致潜在的遇险。例如，若一个加载指令遭受了地址异常，要直到地址转换已经完成的流水线阶段才会发生——通常这时已经晚了。如果下一条指令在取指时（刚好在流水线的开始）就碰到了一个地址问题，那么影响第二条指令的异常事件实际上会先发生。

  为了避免这个问题，早期发现的异常并不立即采取措施；该事件只是被记录并沿着流水线传递。在大多数 CPU 设计中，指定一个特定的流水线阶段作为检测异常的地方。如果当我们的异常记录正在沿着流水线向下传递时，更老指令的后期检测到的事件到达了这个终点线，此时异常记录就直接丢弃。在上面的例子中，取指的地址问题就被忘记——当我们处理完受害指令的问题后重新执行受害指令和后续指令时很可能会再次发生。

- 后续指令无效：因为流水线的原因，处于 **EPC** 中的受害指令之后的指令已经开始了。但是可以向你保证这些指令产生的效果不会影响寄存器和 CPU 状态，而且可以保证这些指令就象异常没有发生一样，根本就不会影响到以后从 **EPC** 重新开始的后续执行过程。

MIPS 实现精确异常的成本非常高，因为其限制了流水线的范围。这在 FPU 中尤其严重/明显，因为浮点操作经常要花费许多个流水线阶段才能完成。跟在 MIPS FP 浮点指令之后的指令不允许提交状态（或者到达它自己的异常决定点），直到硬件能确认浮点指令不会产生异常。

5.1.1 非精确异常——历史上的 MIPS CPU 中的乘法器

MIPS 最初的乘法/除法指令由 **mult** 或 **div** 这样的指令启动/开始，运算需要两个寄存器操作数送入乘法器。程序然后发送一条 **mflo** 指令（有时还要一条

[3] 这不等于说，异常受害指令以及后续指令什么都没干。但是确实有这样的要求：当异常结束后重新执行时，这些指令的行为和异常根本没有发生时的行为完全一样。计算机体系结构师说任何副作用都必须是等幂的 ——做两次和一次的效果一样。

mfhi 指令，如要得到 64 位结果或余数），把结果取回到通用寄存器中。如果计算还没有结束，CPU 就会在 **mflo** 上阻塞。

在 MIPS 实现中，一次乘法要花费 4–10 个周期，但除法可能要 15–30。

通过把这些延时比较长的指令分为两个阶段（“发动”和“获取结果”），指令集鼓励使用和常规整数单元独立的流水线的乘法器。后来的 CPU 提供了更为丰富的指令集，包括乘加指令，给软件提供了利用/发掘流水线的不同方法——详情请参阅第 8.5.5 节。

在符合 MIPS32/64 标准的现代 CPU 上，这些指令的举止规矩。但是老式的 CPU 有可能会出问题。为了简化硬件，最初的体系结构允许乘法/除法运算不可停止，即使就算异常也不能停止。正常情况那也不成问题，但是设想一下我们有如下的代码，其中我们在提取一个乘法单元的结果后立即开始另一条指令：

```
mflo    $8
mult    $9, $10
```

如果发生的异常其重新开始的地址是 **mflo** 指令，那么按照精确异常的规则，第一次执行 **mflo** 指令会被取消掉，寄存器 **$8** 的值就和 **mflo** 指令从未发生过一样。不幸的是，**mult** 指令此时已经开始了，而且既然乘法单元不知道异常就会继续运行。在异常返回之前，计算很可能已经结束了，**mflo** 现在取回的可能是后续的 **mult** 指令的结果。

通过内插至少两条[4]非乘法指令于 **mflo/mfhi** 和 **mult**（或者其它任何可能对 **hi/lo** 写入新值的指令）之间，我们可以避免这个问题（大多数 MIPS32 之前的 CPU 都有这个问题）。

5.2 异常发生的时机

因为既然异常是精确的，从程序员的角度看来异常发生的时间就是没有歧义的：异常之前执行的最后一条指令就是异常受害指令的前一条。如果该异常不是中断，受害指令即是引发异常的指令。

在典型的 MIPS CPU 上中断发生后，在中断处理开始之前完成的最后一条指令就是当检测到中断时正好结束 MEM 阶段的指令。异常受害指令就是那条刚刚结束 ALU 阶段的指令。但是，要小心注意：MIPS 体系结构并不承诺精确的中断延时，中断信号在到达 CPU 核之前可能要花一个或者几个时钟阶段重新同步。

5.3 异常向量：异常处理开始的地方

大多数 CISC 处理器由硬件（或微代码）来分析异常，根据发生的异常类型把 CPU 发送到不同的入口点。甚至连中断都根据哪个中断输入信号激活而在

[4] 为什么是两条？在 MIPS32 之前的 CPU 上两条刚好够避免 **mult** 指令的开始。

不同的入口点处理时，就叫做向量化中断（*vectored interrupts*）。历史上，MIPS CPU 对此做得很少。如果觉得这象是一个严重的疏忽，请考虑下面的因素。

首先，在实践中向量化中断并不象想象的那样有用。在大多数操作系统中，中断处理程序共享代码（为了节省寄存器等原因），CISC 微代码辛辛苦苦花费时间发送到不同的入口点很常见，然后操作系统软件得到一个代号又花更多的时间跳转到公共的处理程序。

其次，很难想象如果不用微代码而完全由硬件能做多少异常分析；在 RISC CPU 上，选用通常的代码就已经足够快。

在此处和别处，你应该对 RISC 这代 CPU 相对其外围设备有多快心里有数。一个有用的中断处理例程要读写一些外设寄存器，在 21 世纪初期的 CPU，一个外部总线周期可能要花费 50–200 个内部时钟周期。很容易在 MIPS CPU 上写出比单个外设访问快的中断处理代码，所以这不大可能成为一个性能瓶颈。2003 年版的 MIPS32 标准中加入的向量化中断选项几乎没人用的事实，进一步证明了这一点。

然而，即使在 MIPS 中，也不是所有异常都是平等的。在体系结构发展过程中也出现了差别。所以我们可以做些区分。

- 用户态地址的 TLB 重填充：在受保护的操作系统中有个特别频繁的异常，与地址转换系统相关（参阅第 6 章）。TLB 硬件只能保存适度规模数量的地址转换，在一个频繁使用的系统运行着一个虚拟存储器的操作系统，应用程序运行到一个 TLB 未记录其地址转换的地址很常见——一个称为 TLB 未命中的事件（因为 TLB 被当作一个由软件管理的高速缓存）。

  当 RISC CPU 刚刚推出的时候，用软件来处理这种情况引发了很多争议，MIPS CPU 为一个选中的 TLB 重填方案提供了重要的支持。硬件提供了足够的帮助使得为选中的重填方案设计的异常处理程序能够在 13 个时钟周期完成。

  作为其中的一部分，常见的一类 TLB 重填被赋予了一个不同于其它异常的入口点，这样精心优化过的重填代码不用再花费时间判断到底发生的是哪种异常。

- 64 位地址空间的 TLB 重填：想要利用 64 位 CPU 上巨大的地址空间的任务，其地址转换次采用略有不同的寄存器布局和另外的 TLB 重填例程；MIPS 称其为 XTLB（我猜“X”代表扩展）。与上面一样，为了让保证高效率用了一个独立的入口点。

- 不经过高速缓存的替代入口点：为了获得异常处理时的良好性能，中断入口点必须位于经过高速缓存的存储区。但是在系统启动的时候不希望这样。复位或者上电后，高速缓存在初始化之前不能使用。如果你需要一个健壮的和能够自我诊断的启动代码，对于引导早期检测到的异常，你只得

用不经过高速缓存的只读存储器的入口点。在 MIPS CPU 里，没有不用高速缓存的“模式”——只有不经过高速缓存的程序地址区域——所以有一个模式位 **SR(BEV)** 用来把异常入口点另行分配到不经高速缓存的、启动安全的 kseg1 区。

- 奇偶/ECC 出错：MIPS32 CPU 可以检测一个数据错误（通常在来自主存的数据中，但是直到在高速缓存中用到才会发现）然后自陷。在高速缓存区来处理高速缓存的错误显然很蠢，所以不管 **SR(BEV)** 状态是什么，*高速缓存出错异常*的入口点总是位于不经过高速缓存的地址空间。

- 复位：对很多目的来说，把复位看作另外一种异常是有道理的，尤其是当许多 CPU 对于*冷复位*（此时 CPU 彻底重新配置；和重新上电不可区分）和 *热启动*（此时软件彻底重新初始化）使用相同的入口点的时候。其实，*不可屏蔽中断 NMI* 的结果相当于热复位的稍弱的版本，区别仅在于它要等待当前指令和尚未完成的存取操作结束才生效。

- 中断：作为 MIPS32（还有 IDT PMC-Sierra 的早期的一些 CPU）的一个可选项，可以设置 CPU 把中断异常发送到单独的入口点。这很方便但很少人用：可能软件作者不能让自己为一个并非普遍通用的特性而让操作作为特例进行处理。

  进一步，在其中某些 CPU 上，你可以使能向量化的中断操作——不同的中断使用多个不同的入口点。这是更具实质性的变化；在本章别的地方讲到过，MIPS 的传统是中断只在软件处理时才有优先级。但当你有两个活动的中断而不得不选择选择一个中断入口点时，硬件必须决定到底哪个优先级更高。因而这个变化对软件的影响要大得多，因为软件丧失了对中断优先级的控制；操作系统维护人员和硬件工程师不得不紧密合作。

所有的异常入口点都位于 MIPS 存储器映像中不作地址转换的区域，不要高速缓存的入口点位于 kseg1 ，需要高速缓存的位于 kseg0。不要高速缓存的入口点当 **SR(BEV)** 置位时是固定的，但是当 **SR(BEV)** 清零时，就可以对 **EBase** 寄存器进行编程来平移所有入口点——一起到——别的内存块。当你的 CPU 是共享 kseg0 存储器的多处理器系统的一部分，但是想要和系统中其它 CPU 分开的单独异常入口点的时候，可以移动中断基址的能力就特别有用。[5]

在这些地址区，表 5.1 给出的 32 位地址通过符号扩展到 64 位内存映像。在 32 位中的程序地址 `0x8000 0000` 和 64 位中的 `0xFFFF FFFF 8000 0000` 相同。

表 5.1 描述了仅有 32 位地址时的入口点——表中的 BASE 代表编程到 **EBase** 寄存器中的异常基址。

最初的异常向量间的距离缺省为 128 字节（0x80）空间，可能是因为最初的 MIPS 体系结构师觉得 32 条指令足够编码基本的异常处理例程了，不浪费太多

[5]EBase 寄存器是在 MIPS32/64 规范的第二版中引入的。

内存而且省下了一条分支指令！现代的程序员很少有这么节省的。

表 5.1: 异常入口点

内存区	入口点	异常处理程在此
片上调试	0xFF20 0200	EJTAG 调试，当映射到“probe”存储区时。参见第 12.1 节关于 EJTAG 片上调试系统的注解。
	0xBFC0 0480	EJTAG 调试，当使用正常 ROM 时。
复位（仅有 ROM）	0xBFC0 0000	复位及不可屏蔽中断入口点。
ROM 入口点	0xBFC0 0000	中断专用——仅当 Cause(IV) 置位时使用，并非所有的 CPU 都有。
	0xBFC0 0380	所有其它异常。
	0xBFC0 0300	高速缓存错误。
	0xBFC0 0200	简单的 TLB 重填（SR(EXL)为零）。
RAM 入口	BASE+0x200+...	多个中断入口点—— VI 模式多七个，EIC 模式 62 个。
	BASE+0x200+...	中断特殊/专用（(Cause(IV)）为一）。
	BASE+0x180	所有其它异常。
	BASE+0x100	高速缓存错误——在 RAM 中但是永远经过不经高速缓存的 kseg1 窗口。
	BASE+0x000	简单的 TLB 重填（SR(EXL)为零）。

下面是 MIPS CPU 决定处理一个异常时所要做的：

1. 设置 EPC 指向重新开始的地址。
2. 设置 SR(EXL) 位，强制 CPU 进入内核态（高特权级）并且禁止中断。
3. 设置 Cause 寄存器这样软件可以看到发生异常的原因。在地址异常时，BadVAddr 也要设置。存储器管理系统异常好药设置某些 MMU 寄存器；第六章会详细讲解。
4. 然后 CPU 开始从异常入口点取指，此后一切交给软件处理。

很短的异常处理例程可以全程运行在 **SR(EXL)** 置位的状态（即我们所说的异常模式），从来不会需要碰到 **SR** 的其余部分。对于更加传统的异常处理程序，保存状态后将控制交给更为复杂的软件，异常特权级提供一个保护伞让系统软件可以在其下安全地保存关键的重要的状态——包括老的 **SR** 值。

加上几个技巧，这个机制可以在基本的 TLB 未中处理程序里允许最小数量的嵌套异常，但是这个问题等我们碰到的时候再仔细讲。

5.4 异常处理：基本过程

所有的 MIPS 异常处理例程都要经过以下相同的阶段：

- 引导：在异常处理程序入口，被中断打断的程序只有很少状态信息保存了下来，所以第一件要做的事就是给你自己腾出足够的空间能够做你想要做的，而且不要覆盖被中断的程序的重要数据。

  这几乎必然是通过用 **k0** 和 **k1** 寄存器（习惯上约定未低级底层异常处理代码所保留）引用一段可以保存其它寄存器的内存空间来实现的。

- 处理不同的异常：查询 **Cause(ExcCode)**，其可能的取值列出在表 3.2 中。它告诉你为什么发生了异常，允许操作系统为不同的异常定义不同的函数。

- 构造异常处理环境：复杂的异常处理程序可能会用高级语言书写，你可能希望能使用标准库函数。你将不得不提供一块堆栈存储区其它软件都没有用到的，保存任何 CPU 寄存器可能对被中断的程序和被调用的例程允许改变的寄存器。

  有些操作系统可能在发送/分派处理不同的异常之前做这个工作。

- 处理异常：现在你想做什么就做什么。

- 准备返回：高层函数通常当作子程序来调用，因而要返回到底层处理代码。在这里，保存的寄存器得到恢复，通过把 **SR** 改回到刚发生异常后的值 CPU 得以返回到其安全（内核模式，异常关闭）状态。

- 从异常返回：异常结束处理是另一个 CPU 状态发生变化的区域，对这部分的描述在接下来的第 5.5 节里。

5.5 从异常返回

控制返回到异常受害指令并从内核态改变到（如果需要）低的特权级必须同时完成（用计算机科学的行话来说，就是“原子的”）。即使你在应用程序中以内和特权级运行一条指令也成为一个安全漏洞；另一方面，以用户特权级试图运行一条内核指令会导致致命的异常。

MIPS CPU 有一条指令 **eret** 完成整个工作；该指令同时清除 **SR(EXL)** 位并将控制返回到 **EPC** 中保存的地址。[6]

[6] 很早期的 MIPS CPU 拥有更复杂的机制，依赖于两级堆栈来保存和恢复异常前 **SR(IE, KU)** 域的值。切换回异常前的模式由一条指令 **rfe** 来完成，必须放在返回到被中断的程序的 **jr** 指令的延迟槽中运行。

5.6 嵌套异常

在许多情况下，你可能想要在你的异常处理例程中允许（或者无法避免）进一步的异常；这称为*嵌套异常*。

如果处理不慎，这可能导致混沌；被中断的程序的要害状态保存在 **EPC** 和 **SR** 中，你必须预计到其它的异常可能会冲掉其值。除出了一个非常特殊的嵌套异常之外，使能其它异常之前都必须保存这些寄存器的内容。此外，一旦重新使能异常，你就不能再依赖为异常处理保留的通用寄存器 **k1** 和 **k2** 了。

一个异常处理程序要想能够活过一个嵌套的异常，必须使用某些内存区域来保存寄存器值。所用的数据结构常常叫做 *异常帧*；嵌套的多个异常帧通常安排在一个堆栈上。

每次异常地要消耗堆栈资源，所以不同容忍任意深度嵌套的异常。大多数系统给每种异常一个优先级，并且安排成当正在处理某个异常时，只允许更高优先级的异常。这样的系统只需要有跟优先级同样个数一样多的异常帧就可以了。

你可以避免一切异常；中断发生了可以用软件单个屏蔽掉以满足优先级规则，用 **SR(IE)** 可以一次性屏蔽全部中断，或者通过（后来的 CPU 才有）异常级位隐式屏蔽单个中断。其它类型的异常可以通过适当的软件约束来避免。例如，在核心态（为大多数异常处理软件所用）不可能发生特权级违反异常，程序可以避免寻址错误和 TLB 未命中异常。当处理高优先级异常时这样做是绝对必要的。

5.7 异常处理例程

下列的 MIPS32 代码片断是简单到不能再简单的异常处理例程了。每次异常时除了递增一个计数器外什么都不做：

```
    .set noreorder
    .set noat
xcptgen:
    la      k0, xcptcount   # get address of counter
    lw      k1, 0(k0)       # load counter
    addu    k1, 1           # increment counter
    sw      k1, 0(k0)       # store counter
    eret                    # return to program
    .set    at
    .set    reorder
```

这个例子看上去没有太大用处：不管什么条件导致的异常，返回后该条件依然存在，所以就会绕圈子来回运行。计数器 xcptcount 最好在 kseg0 中，这样你在读写的时候就不会得到 TLB 未命中的异常。

5.8 中断

MIPS 异常机制是通用的，但是民主地说有两个异常类型发生的次数比其他所有的异常加起来都要多得多。一个是在象 Unix 这样的执行存储器映射的操作座系统上应用程序走出了片上地址转换表的限定的边界；我们以前面提到过这一点，在第六章回过头来再看这个问题。另一个流行的异常就是中断，当 CPU 外部的设备需要注意的时候产生。因为我们要处理的外部世界不会等待我们，所以中断服务时间往往是很关键的。

嵌入式系统的 MIPS 用户应该最关心中断了，这就是为什么要单独用一节专门来讲。我们讲下列几个问题：

- MIPS CPU 中的中断资源：这部分描述你要用到的东西。
- 实现中断优先级：所有中断在 MIPS CPU 中都是平等的，但是在你的系统中，你可能希望对一些比其他中断给与更多的关照。
- 临界区、禁止中断、信号量：常常有必要在某些关键性的操作中间防止中断，但是在 MIPS CPU 上这样做有具体的困难。我们要看看怎样么解决。

5.8.1 MIPS CPU 的中断资源

MIPS CPU 在它的 Cause 寄存器中有一组八个独立的[7]中断位。在大多数 CPU 上，你会发现其中五六个信号来自外部逻辑，而另外两个是纯粹的软件读写的。片上的计数器/定时器（见第 3.3.5 节，由 **Count** 和 **Compare** 寄存器构成）连线到其中的一个；有时也可能计数器/定时器中断何一个外部设备共享一个中断，但是这样做多半是个馊主意。

任何输入信号的电平在每个周期采样一次，如果是能就会导致异常。

CPU 是否愿意相应某个中断受 **SR** 中的位影响。有三个相关的域：

- 全局中断使能位 **SR(IE)** 必须设置为 1，否则不会服务任何中断。
- **SR(EXL)**（异常级）位和 **SR(ERL)**（错误级）位，如果置位会禁止中断（因为紧接异常之后其中之一必然置位）。
- 状态寄存器也有还有八个单个中断屏蔽位 **SR(IM)**，**Cause** 寄存器中的每个中断各一位。每个 **SR(IM)** 要设置成 1 以允许相应的中断，这样程序可以确切知道哪个中断可以发生，哪个不行。

[7]如果在 EIC 模式中就不那么独立，参阅第 5.8.5 节。

软件中断位是干什么用的?

到底 CPU 为什么要在 **Cause** 寄存器中提供两个特殊的位，除非被屏蔽否则一旦置位就立即引发一个中断呢？

答案的关键在于“除非被屏蔽”。这个典型用法是高优先级的中断例程表示将有低优先级的中断例程执行的活动，一旦系统已经处理完所有高优先级的工作的机制。当高优先级的处理结束，软件将打开中断屏蔽，正在等待的软中断就会发生。

没有绝对的原因说为什么同样的效果不可以通过系统软件模拟（比如利用内存中设立的标志）达到，但是软中断位比较方便，因为其以极低的硬件成本嵌入在已有的中断处理机制中。

要发现/揭示/找出当前那个中断正在活动，你要察看 Cause 寄存器的内部。注意到这些恰好是——当前级别——不一定对应于首先引发中断异常的模式。Cause 寄存器活动的输入电平和 SR(IM) 的屏蔽对齐到了同样的位置，以防你万一要对他们进行“与”操作。软件中断位处在最低位，硬件中断依次递增。

用体系结构的术语来讲，所有的中断都是平等的。[8]当处理中断异常时，老式的 CPU 采用“通用的”异常入口点——MIPS32/64 CPU CPU 提供了可选的不同的异常入口点保留给中断使用，这样可以节省几个周期。用 Cause(IV) 寄存器位可以选择这一特性。

中断处理正式开始于你接受到了一个异常并从 **Cause(ExcCode)** 发现该异常是一个硬件中断。通过查询 **Cause(IM)**，我们可以找出哪个中断正在活动也就知道是哪个设备请求服务/发出信号。通常的序列如下：

- 查询 **Cause** 寄存器的 **IP** 域，并和当前中断屏蔽 **SR(IM)** 做逻辑“与”操作，以获得活动的、允许的、使能的中断请求位图。可能不止一个，其中任何一个都有可能是导致中断的原因。
- 选择一个活动的、允许的中断予以关注。大多数操作系统给不同的输入分配固定的优先级，并要求先处理最高的优先级，但这都是软件决定的。
- 你需要在 **SR(IM)** 中保存老的中断屏蔽位，但是你可能已经在主异常例程中已经保存了整个的 **SR** 寄存器。
- 修改 **SR(IM)** 以确保当前中断和你的软件认为具有相同或者较低优先级的所有中断被禁止。
- 如果你没有在主异常例程中作的话，为嵌套的异常处理保存必要的状态（用户寄存器等）。
- 现在修改 CPU 状态以适应中断处理程序的高层部分，在此处有些嵌套的中断或者异常是允许的。

[8]在向量化中断和“EIC 模式”中已经不再如此了，参见第 5.8.5 节，但是很少用到。

在所有情况下，置位全局中断使能位 **SR(IE)** 以允许处理高优先级中断。当你清除异常级的时候，还需要修改 CPU 的特权级域 **SR(KSU)** 以保持 CPU 仍然处于核心态，当然还要对 **SR(EXL)** 清零以离开异常模式，让对状态寄存器所作的修改在别处可以看到。

- 调用中断例程。
- 在返回时，你需要再次禁止中断以便能够恢复中断前的寄存器值并接着执行被中断的任务。要这样做，就需要置位 **SR(EXL)**。但实际中，在进入异常收尾步骤之前把整个 **SR** 寄存器恢复为刚发生异常时的值的过程中，可能已经隐含地置位了 **SR(EXL)**。

当对 SR 做变动时，你需要仔细考虑那些效果会因为流水线的操作而延迟的变动——“CP0 遇险。”有关细节以及怎样编程克服遇险，参见第 3.4 节。

5.8.2 在软件中实现中断优先级

（直到最新的向量化中断功能以前的）MIPS CPU 对于中断优先级有一个简单的办法；一切中断都是平等的。

如果你的系统实现中断优先级策略，那么：

- 在任何时候软件都要维护一个明确定义的中断优先级（IPL），CPU 以该优先级运行。每个中断源分配到其中一个优先级。
- 如果 CPU 处于最低的中断优先级 IPL，则允许任何中断。这是正常程序运行的状态。
- 如果 CPU 处于最高优先级 IPL，则禁止所有中断。

中断处理程序不仅可以按照分配给具体中断源的优先级 IPL 运行，程序员还可以升高和降低 IPL。设备驱动程序应用端与硬件或中断处理程序通信的部分常常需要在临界区防止设备中断，所以程序员会临时升高 IPL 以匹配设备的中断输入位。

在这样一个系统中，在不影响低优先级代码的前提下继续允许高 IPL 的中断，所以我们就有机会对某些中断提供更好的中断响应时间，通常用以换取中断处理程序承诺在短时间内运行完毕。

大多数 UNIX 系统有四到六个 IPL。

虽然也有别的做法，最简单的方案都有以下的性质：

- 固定优先级：在任何 IPL，当前中断以及更低优先级 IPL 的中断被禁止，但是允许更高优先级的中断。对同一 IPL 的不同中断，典型的调度方法是先来先服务。

- IPL 与运行代码的相联系：任意给定的代码片断总是在同样的 IPL 下运行。
- 简单的嵌套调度（在 IPL 0 以上）：除了在最低优先级之外，一没有活动的高优先级中断就返回到被中断的代码。在最低优先级，很有可能有个调度器在不同的任务之间共享 CPU。在一段时间中断活动之后利用这个机会进行重新调度很常见。[9]

在 MIPS CPU 上不同中断级别之间的转换至少伴随着状态寄存器 SR 的改变，因为那个寄存器包含所有的中断控制位。有一些系统上，中断级别的转换需要涉及对外部中断控制硬件进行操作，大多数操作系统有一些全局变量要改变，但此处我们并不关心这些；现在我们要用 **SR** 中断域的一个具体设置来刻画 IPL。

在 MIPS 体系结构中 **SR**（象所有的协处理器寄存器一样）不能直接对单个位清零或置位。因而 IPL 的任何改变都需要一段代码用单独的操作读取、修改、并写回 **SR**：

```
    mfc0    t0, SR
1:
    or      t0, things_to_set
    and     t0, ~(things_to_clear)
2:
    mtc0    t0, SR
    ehb
```

（最后一条指令 **ehb** 是一条遇险防护指令，参见第 3.4 节。）

一般来说，该段代码本身也可能被中断，这样就出了一个新问题：假设我们在标号 1 和 2 之间发生一个中断并且那个中断例程自身引起 **SR** 变化？那么当在标号 2 处写入我们自己修改过的 **SR** 值时，我们就丢失了该中断例程所施加的变化。

事实结果表明，只有在可以保证任何一段具体代码的 IPL 都是常数的（并且运行时 **SR** 不会有其它改变的）系统中，我们能用上面的代码片断——其各种版本在 MIPS 的操作系统实现中相当普遍——而不能会出问题。如果这些条件成立，即使我们在读取-修改-写入序列的中间被打断，也没有什么危害。当中断返回时，将会以和以前同样的 IPL 当然也就是同样的 **SR** 值执行。

在这个假定不成立的地方，我们需要下面的讨论。

5.8.3 原子性以及对 SR 的原子修改

在有多个控制线程的系统中——包括带有中断处理程序的单个应用——你

[9] Linux 系统在中断之后重新调度（即使被中断的任务正处于核心态）称为抢占式的，在 2004 年 v2.6 版本中，抢占式调度已经成为了 Linux 的标准做法。

经常会发现自己在做某件中间不想被半路打断的事情。用更为正式的语言来说将，就是你需要一组修改操作能够原子性的完成，以保证系统中一些合作的任务和中断例程要么看到该组操作全部完成，要么根本一个都没有执行，但是绝不会看到执行到中间的状态。[10]实现原子性修改的代码有时叫做临界区。

在软件运行多个线程的单处理器系统上，一个让当前线程意想不到的线程切换只能是中断的后果。所以在单处理器系统中，通过简单的采用禁止全部中断的办法可以保护任何临界区；这虽然原始但却很有效。

但是正如我们上面所见，这样做有个问题：不能保证禁止中断的步骤（要求对 SR 执行一个读取-修改-写入的顺序操作）本身是原子的。我知道四种解决的办法和一种避免的办法。

如果你知道运行你的软件的所有 CPU 都实现了 MIPS32 第二版，就可以解决这个问题：在这种情况下，你可以用 **di** 指令取代 **mfc0**。**di** 原子性地清零 **SR(IE)** 位，并把原先的 **SR** 返回到一个通用寄存器中。[11] 但是在 MIPS32 第二版的兼容性成为标准配置而非例外配置之前，你可能需要进一步看看别的方法。

一个更通用的做法是坚持不允许中断修改可中断代码的 **SR** 的值；这就要求中断例程永远要在返回前恢复 **SR**，就象要求恢复所有用户级寄存器一样。如果这样的话，上面非原子性的 RMW 序列就没有关系；即使中断插入中间，你用的老的 **SR** 值依然是正确的。这种方法普遍用在 MIPS 上 UNIX 之类的 OS 内核中，并且和每段代码都与固定的 IPL 相联的中断优先级系统相处得很好。

但是有时候这样的限制太多了。例如，当你在每次一个字节的输出端口上发送完最后一个等待的字节时，你可能想在有更多的数据可以发送之前禁止“发送准备好”中断（以避免不停的中断）。还有，有些系统会在不同的中断间轮转优先级以确保中断服务的均匀分布。

另一个解决方案是用一个系统调用来禁止中断（可能你要将该系统调用定义采用单独的置位和清零的参数以相应的更新寄存器）。因为 **syscall** 指令通过引发异常来工作，它能够以原子方式禁止异常。在这种保护下，你的置位和清零操作可以愉快顺利地进行。当系统调用异常处理程序返回时，就会恢复全局中断使能状态（仍然以原子方式）。

系统调用听起来好像是相当重量级的做法，其实不一定花的时间长；但是你得把该系统调用和系统的其它异常分派代码理清楚。

第三个解决方案——所有主要的系统都至少在某些临界区会用到——就是使用下节所述的连锁加载和条件存储指令建立临界区而不用禁止中断。与上面讲的诸种方法不同，这个机制能够正确扩展到多处理器和硬件多线程系统中。

[10]正如一句老话所说的：“决不要给傻子和孩子看半成品”——我按这句话理解，计算机及其软件都是傻孩子。

[11]还有一条使能中断的 **ei** 指令，但是不要用它——用 **di**-返回的 **SR** 值恢复，这样如果你调用 **di** 时碰巧中断已经禁止的话，你的“禁止中断”代码也能正常工作。

5.8.4 允许中断的临界区：MIPS 式的信号量

信号量[12]是实现临界区（当然扩展的信号量可以做更多事情）的一种编程约定。信号量是由并发运行进程共享的内存区，用来安排某些一次只能由一个进程访问的资源。

每个原子执行的代码块都有下面的结构：[13]

```
wait(sem);
/* do your atomic thing */
signal(sem);
```

可以把信号量想成只取两个值：1 意味着“被占用”而 0 意味着“可以用”。`signal()` 简单，就是把信号量设为 0。[14] `wait()` 检查该变量值是否为 0，如不为零则一直等到为 0 才继续。然后将该变量值设为 1 后返回。这个应该不难，但应该看到关键的地方在于检查 `sem` 的值并重新设置的过程本身必须是原子的。高层的原子性（调用 `wait()` 的线程）依赖于能够建成底层的原子性，底层的原子性要能保证“测试且设置”操作在中断面前（或对于多处理器系统，在别的 CPU 访问面前）也能正确运行。

大多数成熟的 CPU 家族有某些特殊的指令用于这个目的：680x0 ——和许多其它的——CPU 都拥有一条原子性的测试且设置指令；x86 CPU 有一条“交换寄存器和内存”操作，可以通过“lock”前缀原子性执行。

对于大型多处理器系统，这种测试且设置的操作变得昂贵；在用户获得信号量完成测试且设置的操作，还有设置的值操作传播到系统中每个高速缓存备份的过程中，基本上所有共享存储器的访问都必须停止。这种办法在大型的多处理器系统上扩展性不好，因为仅仅为了保证能占住偶尔才用到的东西而阻塞了许多可能无关的存储器访问。

如果允许不必事先保证原子性就开始运行“测试且设置”操作，仅当事后知道确实是原子性的时候“设置”操作才能成功的话，那么效率就会高得多。需要告诉软件设置操作是否成功：这样未成功的“测试且设置”操作序列可以隐藏在 wait() 函数中，必要时重试。[15]

这就是 MIPS 有的，依次用 **ll**(load-linked) 和 **sc**(store-conditional) 指令。**sc** 仅在硬件确认自从最近的 **ll** 以来没有竞争的访问时才写入相应的地址位置，并在寄存器中留下一个 1/0 的值表明操作成功或者失败。

[12] 简单的信号量也叫做*互斥*，非正式地称为“锁”。

[13] 这是 Dijkstra 在 1970 年代阐述的提出的概念，类比于铁路信令系统命名为“信号量”。Hoare 将其推广到更广泛的“互相合作的并行进程”环境，把本质上同样的功能称为 `wait()` 和 `signal()` ——就是我们这里用的。Dijkstra 最初的命名为 `p()` 和 `v()`。如果你讲荷兰语的话，就很容易理解为什么叫做“p”和“v”。

[14] 对于操作系统中线程-线程间的信号量，signal() 还要做些工作以“唤醒”在等待该信号量的任一进程。

[15] 当然你最好能够保证不会出现最后永远重试下去的情形——但是如果用其它类型的信号量，你总得要保证等待的任务最后能够执行。

在大多数实现中，这种信息是悲观的：有时即使相应的地址没有被碰过 **sc** 也会失败；如果自从上次 **ll** 后有过中断服务[16] CPU 就让 **sc** 失败，还有大多数多处理器系统在对同一个高速缓存行大小的内存块上的任何写操作都会失败。唯一重要的是当没有竞争访问时 **sc** 通常应当成功，当有的时候保证总是失败。

下面是对二值信号量的 `wait()` 操作：

```
wait:
    la      t0, sem
TryAgain:
    ll      t1, 0(t0)
    bne     t1, zero, WaitForSem
    li      t1, 1
    sc      t1, 0(t0)
    beq     t1, zero, TryAgain
    /* got the semaphore...   */
    jr      ra
```

ll/**sc** 是为多处理器尔发明的。但是即使在单处理器系统上，这种操作也是很有价值的，因为它并不涉及到关闭中断。这种方式避免了上面提到的禁止中断问题，并且可以在减少最坏情形下的中断延时中发挥作用，在嵌入式系统中特别需要。

5.8.5 MIPS32/64 CPU 中的向量化和 EIC 中断

MIPS32 规范的第二版——首先在出自 MIPS 公司的 4KE 和 24K 族处理器 CPU 核心——加入了两个新的特性使得中断处理效率更高。这样省下的成本不多，在大型的操作系统中可能并不重要，但是 MIPS CPU 也用在非常欢迎这种改进的底层的嵌入式环境中。这两个特性就是向量化中断和一种称为*EIC 模式*的可提供大量的不同中断给 CPU 的方法。

如果你启用向量化中断特性，一个中断异常将会根据导致中断的输入信号从八个地址中选择一个开始执行的地址。这里可能会有轻微的模糊性：没有什么东西能够阻止两个中断同时活动，所以硬件就选用没有禁止的活动中断信号中编号最高的。向量化中断通过对 **IntCtl(VS)** 编程来设置，对于不同中断入口点间距给出了几种不同的选择（零值导致所有中断都是用同样的入口点，这就回到了传统做法）。

嵌入式系统常常有大量的中断事件信号，远远超过传统 MIPS CPU 的六个硬件输入。在 EIC 模式，这六个以前相互独立的信号变成一个 6 位的二进制数：零代表没有中断，但是留下了 63 各不同的中断码。在 EIC 模式，每个非零的代码有其自己的中断入口点，允许适当设计的中断控制器能够直接分派 CPU 处理多达 63 个事件。

[16] 更确切地说，如果执行过 **eret** 指令。

向量化的中断（不管是传统的还是 EIC 信令的）在你的系统有一两个中断特别频繁或者对时间要求特别严格的场合下是最为有用的。在更复杂高级的操作系统的约束下，节省下的少量时钟周期很可能因为各种约束而丧失，所以要是发现你最钟爱的操作系统没有用这些特性，请不要感到奇怪。

5.8.6 影子寄存器

即使用中断向量，中断处理例程也还有需要负担避免被中断代码的寄存器的值，必须在做任何有用的工作之前装入自己的地址。

MIPS32/64 规范的第二版允许 CPU 提供一个或个多个不同的通用寄存器组：额外的寄存器组叫做*影子寄存器*。影子寄存器可以用于任何类型的异常，但是对中断最为有用。

一个利用影子寄存器组的中断处理程序不需要保存被中断程序的寄存器值，调用过程中可以在自己的寄存器里保持自身的状态（如果多于一个中断处理程序使用影子寄存器，它们最好就谁用哪个寄存器达成一致）。

一个用了向量化中断和影子寄存器组的中断处理程序的可以从家务活管理解脱出来而运行得飞快/显著的/惊人的/可感觉得到的。但是问题又来了，这个优势可能被操作系统给抵销掉了（特别是，因为操作系统有可能把所有中断禁止一段时间，这段时间可能远远超过我们的超级快的中断处理程序的运行时间）。有些可以从影子寄存器中获益的应用程序使用一个多线程 CPU 也许能更为清楚地收到同样的效果，但是这个话题可就太大了——可以参阅附录 A。

5.9 启动

关于在 CPU 上软件可见的效果，复位几乎和异常完全一样，当然是一种不会返回的异常。在最初的 MIPS 体系结构中，这样做主要是一种经济上的节约实现成本和文档。但是后来的 CPU 已经提供了从冷复位一直到不可屏蔽中断几种不同级别层次的复位。在 MIPS 中，复位和异常条件在不知不觉地互相影响着。

我们要复用常规异常的机制，在复位后 EPC 指向检测到复位时正在执行的指令，大多数寄存器的值都保留了。但是复位破坏了正常的操作，复位发生时正在加载的寄存器或正在写入或重填的高速缓存位置上的值可能被废弃数据填充。

用经过复位仍然保持不变的状态来实现某些有用的生死轮回后的验尸调试是可能的，但是需要你的硬件工程师帮忙。CPU 无法区分复位发生于正在运行的系统还是刚刚上电。死机后的验尸调试留给天才的读者作为作业；我们重点放在系统从头开始启动的情形。

CPU 响应复位时从地址 `0xBFC0 0000` 处开始取指。这是不用高速缓存的 kseg1 区的物理地址 `0x1FC0 0000`。

复位之后，CPU 控制寄存器有足够的状态已经确定，可以让 CPU 执行非高速缓存的指令。“足够的状态”解释为最少刚刚够；注意以下几点：

- SR 中只能保证三样东西：CPU 处于核心态；中断是禁止的；异常被指引到不经高速缓存的入口点——即 **SR(BEV)**=1。在现代的 CPU 上，保证前两个（以及更多）的条件的典型做法就是将异常模式位 **SR(EXL)** 置位，把复位看作异常其实已经隐含了这一层意思。
- 高速缓存处于随机的、无意义的状态，所以从高速缓存的加载可能不读取存储器而直接返回垃圾。
- TLB 处于随机的状态，在初始化之前绝对不能访问（在有些 CPU 上硬件对 TLB 表项重复的可能风险只有最少的保护，结果可能是 TLB 关闭，只能通过再一次复位才能修复）。

传统的启动步骤如下：

1. 分支到主 ROM 中的代码。为什么现在就分支呢？
   - 非高速缓存的异常入口点从 0xBFC0 0100 处开始，并没有给启动代码留下足够的空间可到达“自然的停顿”处——这样反正我们也很快就不得不分支。
   - 分支指令是一个非常简单的测试，可以让我们看到 CPU 是否功能正常并能够成功地读取指令。如果硬件发生了严重的错误，MIPS CPU 很有可能继续按顺序取指（下一个指令极为可能发生永久的异常）。

     如果你用的测试设备能够追踪 CPU 读写的地址，就会显示出 CPU 的复位后的非高速缓存取指；如果 CPU 启动并且分支到了正确的地址，你就有强烈的证据表明 CPU 从 ROM 中读取的数据基本上正确。

     反之，如果你的 ROM 程序直接前进而且弄乱了 **SR**，简单的错误就可能出现各种千奇百怪和难以诊断的结果。
2. 把状态寄存器设置成某种已知的有意义的状态。现在就可以在非高速缓存区可靠地进行加载和存储了。
3. 你可能不得不一直在寄存器中运行，一直到完成一部分 RAM 存储器的初始化（很可能）还有对其完整性的一个快速检查。这个过程可能会很慢（我们依然在非高速缓存区的 ROM 中运行），所以你可能要把你的初始化和检查限制到一块能够容纳下 ROM 程序的数据的内存区域内。
4. 你可能还不得不和外部世界（一个控制台端口或者诊断寄存器）作些联系，以便能够报告初始化过程中的问题。

5. 现在你可以给自己分配一些栈空间，并设置好足够多的寄存器，以便能够调用标准的 C 例程。

6. 你现在可以初始化高速缓存然后舒服的运行。有些系统可以从高速缓存运行 ROM 代码而有些不能；在大多数 MIPS CPU 上，向高速缓存提供数据的存储器必须能够提供 4/8 个字的突发传输，你的 ROM 子系统不一定能满足。

5.9.1 检测和识别你的 CPU

你可以从 **PRId(Imp)** 和 **PRId(Rev)** 域来识别辨认你的 CPU 的实现号和厂商定义的版本修订号。但是最好尽可能少依赖这些信息；如果你依赖 **PRId**，可以肯定即使将来的 CPU 没有任何会给程序带来麻烦的新特性，你也将不得为新 CPU 修改你的程序。

只要你能直接检测某个特性，就那样做吧！在不能直接进行可靠检测的地方，对于 MIPS32/64 兼容的 CPU，3.3.7 节描述的各个 **Config** 寄存器中包含有大量有用的信息。只有在用尽了其它所有方法之后都不行，这时才应该考虑 **PRId**。

不过，引导 ROM 和诊断软件肯定要以某种易读的形式显示 **PRId**。如果你不得不包含一个实在令人不快的软件修正用来克服硬件缺陷的话，好的做法是让代码根据 **PRId** 条件执行，这样你的代码在将来修正后的硬件中就不会用到了。

既然我们推荐你检测各个特性，在这里给出一些怎样去做的例子：

- 我们有浮点硬件吗？你的 MIPS32/64 CPU 应当通过 **Config1(FP)** 的寄存器位告诉你。对于老式的 CPU，一种推荐的技巧是置位 **SR(CU1)** 使能协处理器 1 操作，并用一条 **cfc1** 指令从协处理器 1 寄存器 0 获得其中容纳的版本号 ID。**ProcessorID** 域（位 8–15）为非零值表明存在 FPU 硬件——经常会在 **PRId(Imp)** 域也看到同样的值。一个持怀疑态度还不放心的程序员[17]可能还会接着检查是否能从 FPU 寄存器存入和读取数据。除非你的系统支持无条件使用浮点单元，否则别忘记完了后复位 **SR(CU1)**。

- 高速缓存大小：在一个符合 MIPS32/64 的 CPU 上，依赖于配置寄存器 **Config1-2** 中的编码的值可能会更好一些。但是你也可以通过读写高速缓存标签查看什么时候发生回绕来推算出高速缓存大小。

- 我们有 TLB 吗？这是内存地址转换硬件。**Config(MT)** 位告诉你是否有。但你也可以读写 **Index** 的值或者寻找连续计数的 **Random** 寄存器存在的

[17]我假设就是你，亲爱的读者。

证据。如果看上去有戏，你也许想要检查是否能够对 TLB 表项数据进行存取。

- CPU 时钟频率：设法找出时钟频率常常是很有用的。你可以通过运行一段已知长度的、位于高速缓存内的、需要花费固定数量的 CPU 周期的循环代码，然后和 CPU 外部已知速度的计数器在执行前后的值进行比较。一定要保证你真的是在运行高速缓存中的，否则就会得到奇怪的结果——记住记得有些硬件不能在高速缓存中运行 ROM。

  Linux 内核在引导时执行这项工作并报告一个数，叫做 BogoMIPS，以强调该数值和 CPU 性能之间的任何关系都是假的。

  如果你的 CPU 符合 MIPS32/64 第二版（2003 年的规范），可用 **rdhwr** 指令让你直接访问主 CPU 时钟和 **Count** 寄存器递增速度的比例因子。比较计数器速度和外部参考标准要容易一些。

如果你给出 CPU ID、时钟频率、高速缓存规格信息，比如说作为启动信息的一部分显示出来，有一天某个维护工程师会感谢你的。

5.9.2 引导步骤

启动代码受两个相反但是都想要的目标之间的冲突的折磨。一方面，对于硬件完整性应当作最少的假定，试图在使用每个子系统之前都要检查（想象一下爬梯子的时候，在每一级踏稳之前要试一下才踏上去）。另一方面，尽量减少复杂难懂的汇编代码。引导程序几乎从来都不是关心性能的，所以很想要早点切换到高级语言。但是高级语言代码倾向于要求已经有更多的子系统能够运行。

当你处理完 MIPS 具体的障碍（比如设置 **SR** 以便至少能够进行存取）后，主要的问题就是过多久才能够给程序提供可读可写的内存，这对于调用 C 语言写的函数是必不可少的。

有时候诊断套件包含一些像原来的 PC BIOS 中依次测试每一条 8086 指令的代码那样奇怪的东西。在我看来这好像是把你的自行车和车子自身用链子锁到一起来防止被盗。如果任何子系统和 CPU 在同一块芯片中实现，免去对该子系统的检查不会有太大损失。

5.9.3 启动应用程序

要想能够启动一个 C 语言的应用程序（可能其指令来源于 ROM），你需要有可以写入的存储器，是因为以下三个原因。

首先，你需要堆栈空间。分配足够大的一块可写的内存并且把 **sp** 寄存器初始化为该块存储器地址的上界（对齐到八字节的边界）。算出堆栈应该多大是很困难的：计算时多留点余地会更健壮。

然后，你可能需要初始化一些数据。正常情况下，C 的数据区由程序装载器初始化，以设置被赋予数值的变量。大多数能够用于嵌入式系统的编译系统允许只读数据在目标代码中由一个单独的段管理，并且放入 ROM 存储器。

只有在编译系统和运行系统相互协调安排在调用 main() 之前把可写数据的初始化代码从 ROM 复制到 RAM 的条件下，才能用初始化了的可写数据。

最后，C 程序对所有没有明确初始化的 `static` 和 `extern` 数据项用一段单独的内存块——这块内存区域有时候称为 BSS，称为 BSS 的原因由于年久已经失传了。这种变量应当清零，通过在程序的开始前将整个数据区清零就很容易做到这点。

如果你的程序是仔细构建的，这些就够了。但是，问题可以搞得更复杂：要注意你的 MIPS 程序构建时没有采用全局指针寄存器 **gp** 来加速对非堆栈变量的存取，否则还需要做更多的初始化。

5.10 指令仿真

有时候异常被用来调用一个代替异常受害指令的软件处理程序，就象在不支持浮点硬件的 CPU 上使用软件来实现浮点操作那样。调试器和其它的系统工具有时也可能想这样做。

要仿真一条指令，你需要找出指令、解码、并找出它的操作数。MIPS 指令的操作数在寄存器中，到这个时候这些异常前的寄存器值已经被放到异常帧中去了。有了这些值作为武器，你可在软件中执行操作，把结果以打补丁的方式存入相应结果寄存器在异常帧中的拷贝。然后需要对存储的异常返回地址作些小调整这样就跳过被仿真的指令然后返回。我们将一步一步地走一遍这个过程。

找出导致异常的指令容易；通常就是 **EPC** 指向的那条，除非处于分支延迟槽内，此时 **Cause(BD)** 置位异常受害指令位于地址 **EPC+4** 处。

要解码指令，你需要某种反汇编表。作为广为使用的 GNU 调试器 gdb 中被用来产生反汇编列表的一部分，有个面向解码的 MIPS 指令列表。只要 GNU 的授权条件对你没有问题，就可以用它帮你节省时间和精力。

要找出操作数，你需要知道异常帧的位置和布局，这个取决于你的具体操作系统（或者异常处理软件，如果不好意思叫做操作系统的话）。

你不得不自己想办法来执行这个操作，这里也还需要能够取得异常帧，以把结果正确放回到相应寄存器被保存的拷贝中去。

当增加存储的 **EPC** 值以越过被仿真的指令时不小心就会落入陷阱。如果被仿真的指令位于分支延迟槽中，你不能就简单返回到“下一条”指令——那就等于没有发生分支，而事实上你不知道是否发生分支。

所以当被仿真的目标指令位于一个分支延迟槽时，还要仿真分支指令，测试看看发生了分支还是没有。如果发生了分支，需要计算分支目标然后直接从异常返回到那里。

所幸的是，所有的 MIPS 分支指令都很简单的，而且没有副作用。

第 6 章　底层内存管理与 TLB

我们倾向于直接从最底层引入本书中的大部分主题进行探讨，对于一本关注计算机底层体系结构的书而言，这似乎是自然而然的。然而——除非是你已经对虚拟存储系统和操作系统怎样使用虚拟存储系统非常熟悉——建议你现在把书翻到第 14.4 节，看看该节讲述的 Linux 中的虚拟存储管理机制。一旦你对此搞清楚了，再翻回到这里来看看硬件的细节，看看相同的硬件在其它环境下如何工作。

当我们自顶向下思考的时候，硬件常常被称为存储器管理单元或者 MMU。当我们自底向上的看的时候，我们主要集中于主要的硬件 TLB（全称是“translation lookaside buffer(转换旁视缓冲器)”，不过这个全称对我们理解没有多大用处）。

6.1　TLB/MMU 硬件及其作用

TLB 是把你的程序用的地址（程序地址或者虚拟地址）转换成为访问存储器的物理地址的硬件。操作系统对于内存转换的控制是所有软件安全特性的关键。

在 MIPS CPU（以及所有的现代 CPU 上）上，地址转换以 4KB[1] 大小为单位，称为页。页内的 12 位地址只是从虚拟地址简单地传递到物理地址。

转换表中每一项含有一个页的虚拟地址（VPN 即虚拟页号）和一个物理页地址（PFN 代表页帧号）。当程序给出一个虚拟地址时，该地址和 TLB 中的每个 VPN 做比较，如果和某一项匹配就给出相应的 PFN。TLB 是一种称之为相联存储器或者内容寻址的存储器——不是按照索引来选择而是根据内容来选择某一项。这逻辑是相当复杂的，其中每一项都有内建的比较器，复杂度和性能扩展性很差。所以典型的 TLB 只有 16 到 64 项。

有一组标志位和每个 PFN 一起存储并一起返回，标志位让操作系统可以指定某一页为只读或者指定某页的数据是否可以高速缓存。

[1] MIPS 硬件也支持大于 4KB 的页，而且内存这么便宜，使用更大的基本页尺寸对软件也有利，但是已经形成的 4KB 习惯很难摆脱。特殊的地址转换偶尔用到大得多的页。

大多数现代的 MIPS CPU（以及所有的 MIPS32/64 CPU）采用双倍存储，每一个 TLB 项容纳一对相邻的虚拟页面对应的两个单独的物理地址。

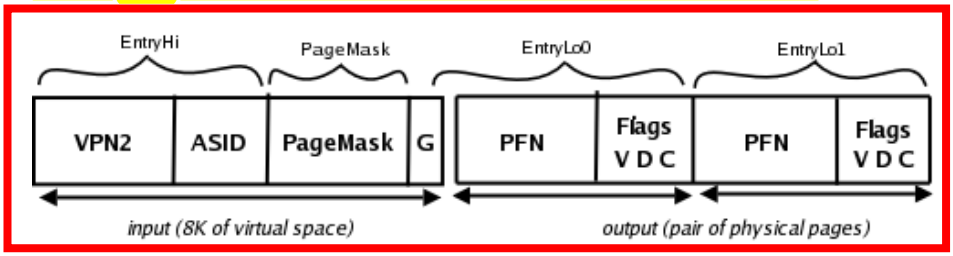

图 6.1: TLB 数据项

图 6.1 给出了一个 TLB 项。每个域标上软件加载和读取 TLB 表项的时候用的 CP0 寄存器的名字，下一节讲这些寄存器。

当你在运行一个真正的复杂的操作系统时，软件很快就会覆盖比 TLB 的转换表能容纳的更多地址。这可以通过把 TLB 当作一个软件管理的最近用过的地址转换的高速缓存来维护得到解决。当一个需要的地址转换不在 TLB 中时，就会产生一个异常，异常处理程序算出并安装正确的地址转换。很难相信这种做法会有效率，但是事实确实如此。

6.2 TLB/MMU 寄存器

象 MIPS CPU 中的其它东西一样，MMU 控制也要受到很少几条额外指令和协处理器 0 的一组寄存器的影响。表 6.2 列出了这些控制寄存器，我们在第 6.3 节会讲到相关指令。所有 TLB 的表项写入或者读取都要通过寄存器 **EntryHi**、**EntryLo0–1** 和 **PageMask** 来进行。

6.2.1 TLB 关键字域——EntryHi 和 PageMask

图 6.2 给出了这两个寄存器，和 **EntryLo**（如下所述）一起是 TLB 表项对程序员唯一可见的视图。图中同时给出了 MIPS32 和 MIPS64 版本的 **EntryHi** 以及下列的域：

- VPN2（虚拟页号）：这部分是程序地址的高序位（低位的页内地址 0–13 位略去了）。0–12 位属于页内地址，但是程序地址的第 13 位也不参与查找：每项映射一对 4-KB 大小的页，位 13 自动在两个可能的输出值之间进行选择。

  在重填异常发生之后，自动设置该域匹配未能转换的程序地址。当你想要写入另外的 TLB 表项或试图检测 TLB 时，就只能手工设置。

  EntryHi 的 MIPS64 版本允许实现比到目前为止更大的虚拟地址区域——达到 62 位，然而当前的典型 CPU 只能实现 40 位。VPN2 实现的

表 6.1: 用于内存管理的 CPU 控制寄存器

寄存器助记符	CP0寄存器号	描述
EntryHi	10	这些寄存器一起构成了一个 TLB 表项所需的一切。所有对 TLB 的读写都要经过它们。EntryHi 存有 VPN 和 ASID；MIPS32/64 CPU 每一项映射到两个连续的 VPN 到不同的物理页，这样两个页的 PFN 和存取权限标志分别由名字为 EntryLo0 和 EntryLo1 的寄存器单独指定。 EntryHi(ASID) 域具有双重职责，因为它要记住当前活动的 ASID。 EntryHi 在 64 位 CPU 中增长到 64 位，但同时对不要长地址的软件保持 32 位布局不变。 PageMask 可以用来创建映射到大于 4 KB 的页的项；参见第 6.2.1 节。有些 CPU 支持更小的页，但在本书很少提到。
EntryLo0–1	2–3	
PageMask	5	
Index	0	这个值决定相应指令要读写的是哪一个 TLB 表项。
Random	1	这个伪随机数值（实际上是一个自由运行的计数器）被 tlbwr 指令用来将一个新的 TLB 表项写入到一个随机选择的位置去。当为喜欢随机替换的软件（可能没有别的替代方案）处理 TLB 重填时可以节省时间。
Context	4	这些是提供方便的寄存器，用来加速 TLB 重填自陷的处理。高位是可读写的，低位取自于未能转换的地址的 VPN。 这些寄存器域的布局可以保证，如果对存储器转换记录的驻留内存的副本根据你的意愿进行安排，那么在一次 TLB 重填自陷后，**Context** 就会包含一个指向映射相应出错地址所需要的页表记录的指针。参见第 6.2.4 节。 XContext 针对超过 32 位有效地址空间的进程发生的自陷完成同样的工作；简单的直接把 Context 的布局扩充到更大的地址空间是不行的，因为结果会导致太大的数据结构。有些 64 位的软件对于 32 位的虚拟地址空间没有意见；但是如果不够用，64 位 CPU 装备有“模式位” **SR(UX)** 和 **SR(KX)**，通过设置可以调用另外的 TLB 重填处理程序；反过来该处理程序又能用 **XContext** 来支持一个巨大但仍然可以管理的页表格式。
XContext	20	

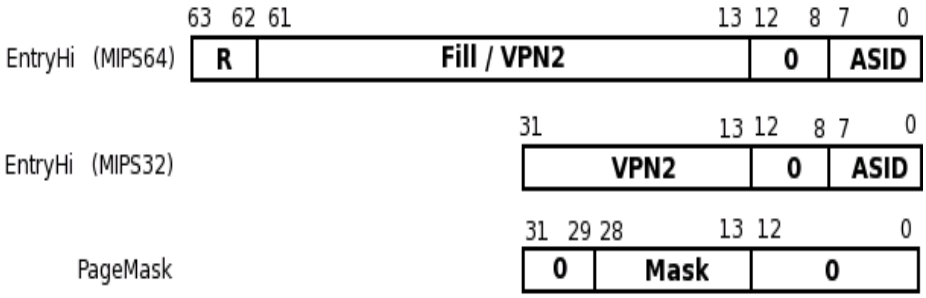

图 6.2: EntryHi 和 PageMask 寄存器域

位向上增长远远足以对付，实际上，你要从这个寄存器中找出你有多少虚拟空间。如果你写入全 1 的值到 **EntryHi(Fill/VPN2)** 然后重新读回，**EntryHi(VPN2)** 的有效位就是那些读回时值为 1 的位。

VPN2 中超过 CPU 实际使用的高位地址必须写入全 0 或者全 1，和 **R** 域的最高有效位要匹配。这等于说，当访问核心专用地址空间时高位为全 1，否则为全 0。

如果你仅仅使用 32 位的指令集，这一切就会自动发生，因为当你这样工作时所有的寄存器包含的值都是一个 32 位数的 64 位有符号扩展。这样一来，运行在 64 位硬件之上的 32 位软件可以认为硬件只有 32 位的 **EntryHi** 结构。

还要注意到，在 **VPN2** 域以下还有几个未用的零位；实际上这些位并不是永远都不用的——有些 CPU 配置可以支持 1-KB 大小的页，这种情形 **VPN2** 及相应的屏蔽位向下延伸两位。

- **ASID**（地址空间标识符）：正常情况下，这部分留下来保存操作系统当前地址空间的标识。异常不会影响该域，所以重填异常之后，该域对当前运行的进程来说仍然是正确的。

  使用多个地址空间的操作系统会维护该域为当前地址空间。但是因为该域被放进了 **EntryHi**，用 **tlbr** 来检查 TLB 表项的时候不得不十分小心；该操作会重写整个 **EntryHi**，执行后必须恢复正确的当前 ASID 的值。

- R:（64 位版本才有）这是一个地址区域选择符。你可以一致地把该域看作是 **EntryHi(VPN2)** 的更多位；只不过是 64 位 MIPS 虚拟地址的最高位。但是如果你还记得 64 位的扩展存储器映射（参见第 2.8 节的图 2.2），可以看到这些高位选择具有不同访问权限的存储区域。

R 值	区域名	简短描述
0	xuseg	用户态可访问的虚拟存储器低地址区
1	xsseg	管理态可访问的空间（管理态是可选的）
2	xkphys	核心专用的大的物理存储器（用不用高速缓存都包括）窗口
3	xkseg	核心态空间（包括 MIPS32 兼容段）

R 位和 **VPN2** 的高位并不同，因为它们的确可以有不同取值——由具体实现定义的 **EntryHi(VPN2)** 的高多少位只能取全零或者全一。

PageMask 寄存器允许你设置 TLB 域来映射更大的页。**PageMask(Mask)** 域代表 TLB 表项的一部分，值为 1 的位使得虚拟地址的对应位在执行 TLB 表项匹配时被忽略（导致相应位原封不动直接传递到结果的物理地址），等效于匹配一个更大的页尺寸。

没有 MIPS CPU 能够在 **PageMask(Mask)** 域允许任意的位模式。大多数允许的页大小在 4 KB 和 16 MB 之间以四倍递增。

24–21	20–17	16–13	页大小
0000	0000	0000	4 KB
0000	0000	0011	16 KB
0000	0000	1111	64 KB
0000	0011	1111	256 KB
0000	1111	1111	1 MB
0011	1111	1111	4 MB
1111	1111	1111	16 MB

至少有一个老式的 MIPS CPU 仅支持 4-KB 和 16-MB 的页，但仍用了这些尺寸的标准编码；建议你做一些检测来看看 **PageMask** 中哪些值是“顽固”的。

如果你的 CPU 能够支持 1-KB 大小的页，在 **PageMask** 的底部还要有两个额外的位，对它的设置遵循同样的模式，我们把它留给读者作为练习。

6.2.2 TLB 输出域——EntryLo0–1

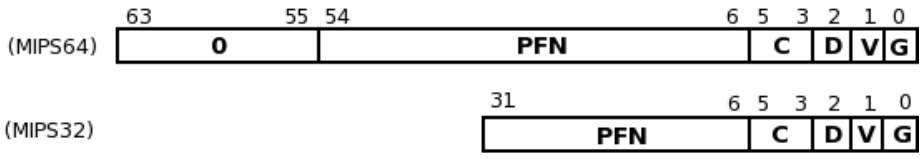

图 6.3: EntryLo0–1 寄存器域

图 6.3 给出了 **EntryLo** 的 MIPS32 和 MIP64 版本，它们的各个域如下：

- PFN: 这部分是和 **EntryHi(VPN2)** 域对应的地址转换项的物理地址的高位部分。该域的有效宽度取决于你的 CPU 所支持的物理存储空间的大小。MIPS32 CPU 配的外部接口经常限制到 2^{32} 字节的范围，但是 MIPS32 版本的 **EntryLo** 可以潜在的支持多达 2^{38} 字节的物理范围（26 位的 PFN 提供了2^{26} 个页面，每个大小为 4 KB 或者 2^{12} 字节）。

- C: 一个 3 位的域，本来是为高速缓存一致性的多处理器系统定义的，用来设置“高速缓存算法”（也叫做“高速缓存一致性属性(cache coherence attribute)” ——有些手册把该域叫做“CCA”）。典型的操作系统都知道有些页面不需要在多个高速缓存之间自动跟踪变化——已知只有一个 CPU 使用的页面，或已知为只读的页面，这些不需要太多的关心。关闭对这些页面存取的高速缓存侦听和交互可以让系统更加有效率，这个域由操作系统用来记录相应页为，比如说，可以高速缓存但是不需要一致性管理（“cacheable noncoherent”）。

  但是这个域也在瞄准嵌入式应用的 CPU 中使用，这时用来选择高速缓存工作方式——例如，标记某个具体的页为“透写”式管理（就是说，所有在那里进行的写操作都同时直接送到主存和高速缓存的映像）。

  该域仅有的普遍支持的值包括，“不用高速缓存(uncached)”(2) 和 “可高速缓存但不要一致性(cacheable noncoherent)”(3)。

- D(dirty): 这个位作为写允许位。置位为 1 允许写入，0 导致用该地址转换的写入操作发生自陷。参阅第 14.4.7 节对 *脏(dirty)* 这个术语的解释。

- V(valid): 若为 0 则对与该项匹配的地址的任何使用都导致异常。要么用来标记一个不可访问的页（在真正的虚拟存储系统上），要么就是标记一对地址转换之一的 **EntryLo** 部分为不可访问。

- G(global): 当 TLB 表项的 G 位置位时，TLB 项将仅仅在 **VPN** 域上匹配，而不管 TLB 项的 ASID 域是否匹配 **EntryHi** 中的值。这样提供了一种有效的机制来实现所有进程共享的地址空间部分。注意尽管有两个 **EntryLo** 寄存器，在 TLB 数据项中却只有一位 G 位：如果你的 **EntryLo0(G)** 和 **EntryLo1(G)** 不同，那就会坏事。

- 叫做 0 的域和未用的 PFN 位: 这些域总是返回零，但是和许多保留域不同，写入的时候不一定要写入零（不管写入什么值都没用）。这一点很重要；意味着在重填 TLB 时内存驻留的用来产生 **EntryLo** 的数据可以在这些域包含一些由软件自己解释的数据，TLB 硬件忽略这些域，而不用浪费宝贵的 CPU 周期去屏蔽它。

6.2.3 选择 TLB 的表项——Index、Random 和 Wired 寄存器

Index 寄存器用来挑选一个具体的 TLB 表项——取值从零一直到表项的总数减一。当你有意想要读写一个具体数据项的时候，需要设置 **Index**，当你用 **tlbp** 用软件搜索一个 TLB 表项时 **Index** 也会自动设置。

Index 的低位保存 TLB 的索引。[2]这不需要很多位，因为还没有 MIPS CPU 有超过 128 项的 TLB。最高位（位 31）有特殊意义，当检测未能找到匹配项的时候由 **tlbp** 置位。位 31 选得好——因为这样看起来象负值，很容易测试。

Random 保存到 TLB 的一个索引，CPU 每执行一条指令就计数一次（向下递减计数，如果这点对你重要的话）。该值为写数据项指令 **tlbwr** 充当到 TLB 的索引，在需要一条新的 TLB 项的时候，帮助实现随机替换策略。

正常使用时，你永远不必读写 **Random** 寄存器。硬件在复位时把 **Random** 域设置为最大值——TLB 表项的最高编号，每个时钟周期递减计数，递减到最小值后又回绕到最大值，如此周而复始。

TLB 中从 0 开始向上，凡是索引小于下限值的表项不受随机替换的影响，操作系统可以将这些槽用于永久转换地址项——在 MIPS OS 的文档中称为 禁锢的*(wired)*。**Wired** 寄存器允许你指定下限值因而可以决定禁锢的转换项个数。当你写入 **Wired** 寄存器时，**Random** 寄存器被复位到 TLB 的顶部。

6.2.4 页表存取辅助寄存器—— Context 和 XContext

当因为 TLB 中不存在地址转换而导致 CPU 发生异常时，未能转换的虚拟地址已经在 **BadVAddr** 了。页级地址也反映在 **EntryHi(VPN2)** 中，在那里被预设到一个值，该值恰好就是为转换出事地址创建新的数据项所需要的值。

但是为了进一步加速对这个异常的处理，**Context** 或者 **XContext** 寄存器用可直接作为指向基于内存的页表的指针的格式对页级地址重新打包。

MIPS32 CPU 只有一个 **Context** 寄存器，能够帮助 32 位虚拟地址空间的 TLB 重填过程；MIPS64 CPU 增加了 **XContext** 寄存器，当使用更大的地址空间（可达 40 位）时要用到。

这些寄存器（包括 MIP32 和 MIP64 版本）如图 6.4 所示。

注意到 **XContext** 是唯一的一个 MIPS64 定义中没有确切定义内部各域边界的寄存器：**XContext(BadVPN2)** 域在支持超过 40 位虚拟地址空间的 CPU 上自动增长，并且将 **R** 和 **PTEBase** 域向左推移（要保证能放得下，就要自动缩小后者）。

第 6.2.1 节描述了图 6.2，并解释了怎样找出你的 CPU 用了多少位。各个域如下：

[2]古老的 R2000 兼容的 CPU 的寄存器布局与此不同，是从第 8 位开始的。你不大可能再碰到遇到这种老古董。

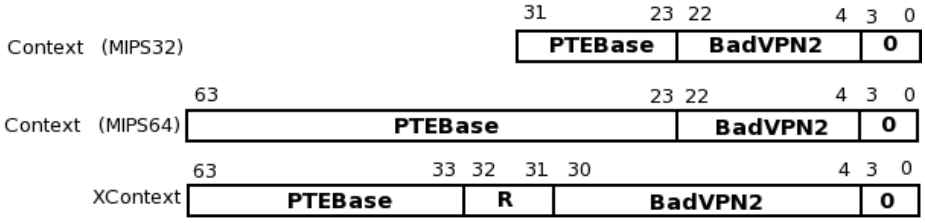

图 6.4: MIPS32 和 MIPS64 的 Context 寄存器的域

- Context(PTEBase): 该域只是保存你放进去的值。为了能够充分按照设计者意图利用寄存器，写入一个驻留内存页表的起始地址的（适当对齐的）高位。该起始地址必须选得在第 22 位及以下为零——即 8-MB 的边界。虽然在物理存储器或者不做映射的存储器中提供对齐会让总体效率降低，其目的是这个表应该能放在 kseg2 映射的地址区。具体怎样工作见下。

- Context(BadVPN2)/XContext(BadVPN2): 在 TLB 相关的异常之后，该处保存页地址，正好就是 **BadVAddr** 的高位。

  为什么名字中有个 2 呢？还记得，在 MIPS32/64 TLB 中，每项映射一对相邻的虚拟地址页面到两个独立的物理页面。

  BadVPN2 的值从位 4 开始，这样可以预先计算一个指针，指到一个基址在 **PTEBase** 中，每项占 16 字节的表里头。如果操作系统对这个表进行维护，使得一个具体的虚拟页地址隐含访问的表项包含的数据恰好就是创建该页地址转换的 TLB 表项所要的正确的 **EntryLo0–1** 数据，那么 TLB 未命中异常处理程序要做的工作就可以降到最少；在 6.5 节就可以看到这一点。如果你只是转换 32 位的地址而且不需要在页表中保存太多的软件专用的状态位，那么可以用八个字节的页表项。这最后成为 Linux 为什么不以上述方式使用 **Context** 寄存器的原因之一。如果你的 CPU 可以处理超过 40 位的用户虚拟地址空间，**XContext(BadVPN2)** 域可能比图 6.4 中所示的要大。这时，**R** 和 **PTEBase** 被压缩以便腾出空间。

- XContext(PTEBase): 64 位地址区域的页表基址对齐要求所有低于 **XContext(PTBase)** 的位都为零，就是说要对齐到 8GB 的边界。这个要求听起来很过分，但是基本的 MIPS64 存储器映射本来就有一个充分大的、映射的、内核专用的区域（“xkseg”）。

- XContext(R): TLB 未命中可以来自 CPU 存储器映射的任何区域，不仅限于用户空间。所有的区域位于一个 64-位空间之下，但是远远小于要装下全部地址的需求（你可能想要参考 2.8 节图 2.2 ）。可用的 64 位虚拟地址空间被分为四个“xk...”段，在每一段中可以使用 62 位段内地址。

  如这里所示，未命中的地址保存为一个单独的 40 位的区内页地址（**BadVPN2**）

和一个 2 位的映射区域选择符 **XContext(R)**，这样仅仅是为了节省 **XContext** 的空间。选择符定义如下：

R 值	区域名	简短描述
0	xuseg	虚拟存储空间底部用户态可访问的空间
1	xsseg	管理态可访问的空间（管理态为可选的）
2	xkphys	对应于不做地址置映射（地址转换）的段，未用
3	xkseg	核心态映射的空间（包括老的 kseg2）

注意到并非所有的操作系统都象最初设想的那样使用 **Context/XContext**——特别值得一提的是，Linux 没有这样做。我们以后讨论其原因。

6.3 TLB/MMU 的控制指令

指令：

```
tlbr    # read TLB entry at index
tlbwi   # write TLB entry at index
```

在 **Index** 寄存器选择的 TLB 表项和 **EntryHi** 和 **EntryLo0–1** 寄存器之间传送 MMU 数据。

你不会经常地读取 TLB 表项；当确实这样做的时候，记住你已经重写了 **EntryHi(ASID)** 域，而该域应当由操作系统维护用来选择当前进程的地址空间。所以还得把原来的值放回去。

指令：

```
tlbwr   # write TLB entry selected by Random
```

把 **EntryHi**（当然包括其 ASID 域）、**EnbtryLo** 以及 **PageMask** 的内容拷贝到由 **Random** 寄存器索引的 TLB 表项中去——如果你采用随机替换策略那么这样会节省时间。实践中，在 TLB 重填异常处理程序里使用 **tlbwr** 来写入新的 TLB 表项，但是在其它地方都使用 **tlbwi**。

指令：

```
tlbp    # TLB lookup
```

搜索虚拟页号及 ASID 跟当前 **EntryHi** 中的值相匹配的 TLB 项，并把该项的索引保存到 **Index** 寄存器。若没有找到匹配则 **Index** 寄存器的第 31 位置位——这样看上去像个负值，因而易于检测。

如果多于一项的值发生匹配，则可能出任何结果。这是一个不该发生的严重错误：软件应当极为仔细地设计过，保证绝对不会对任何地址安装第二次地址转换。

注意 **tlbp** 并不取得 TLB 的数据；如果需要的话，随后再运行一条 **tlbr** 指令。

在大多数 CPU 上，TLB 内部是流水线的这样地址转换可以高效进行。象上面这样的管理/诊断操作可能无法适应标准的流水线流程，所以如果在 TLB 维护指令之后太快就运行使用转换地址的指令就可能会出现遇险。参见第 3.4 节关于 CP0 遇险的内容，但是为了避免难以处理的冷僻情形，通常用在 kseg0 之类的非转换区域执行的软件进行 TLB 维护。

6.4 TLB 编程

对 TLB 表项的设置是这样的：写入 **EntryHi** 和 **EntryLo** 中必要的域，然后用一条 **tlbwr** 或者 **tlbwi** 指令传送到适当的 TLB 表项。

当处理 TLB 重填异常时，你会发现 **EntryHi** 寄存器已经由异常设置好了。

要特别小心不要生成和同一程序地址/ASID 对相匹配的两个 TLB 表项。如果 TLB 包含重复的数据项，当这样一个地址进行转换或检测时，有损坏 CPU 芯片的潜在可能。有些 CPU 在这种情况下通过关闭 TLB 来执行自我保护，此时 **SR(TS)** 置位。此后 TLB 将停止匹配，直到一次硬件复位才能恢复。

系统软件常常根本不需要读取 TLB 项。但是如果你需要读取，你可以用 **tlbp** 来查找和某个具体程序地址相匹配的 TLB 项来设置 **Index** 寄存器。不要忘记保存 **EntryHi** 并在事后恢复——**EnbtryHi(ASID)** 域可能很重要。

用一条 **tlbr** 指令把 TLB 的数据项读入到 **EntryHi** 和 **EntryLo0–1**。

在 CPU 文档中会看到有些地方提到单独的“ITLB”和“DTLB”（有时统称“uTLB” ——“u”代表“micro”）结构。这个 micro-TLB 对指令和数据分别进行地址转换；是一个微小的硬件管理的地址转换的高速缓存，其操作对软件完全透明——在你对主 TLB 写入一项的时候就会自动作废。

6.4.1 重填是怎样发生的？

当程序访问一个要进行地址转换的地址区域（通常是保护的 OS 下的应用程序的 kuseg 和核心特权级映射下的 kseg2），而发现没有地址转换记录时，CPU 就会产生一个 TLB 重填异常。

TLB 只能映射当代服务器或者工作站的物理存储器范围的一部分。大型操作系统需要维护能够容纳大量页地址转换的驻留内存的页表，用 TLB 作为最近用过的地址转换的高速缓存。为了提高效率，通常把页表组织成一些直接可用的 TLB 数据项组成的数组；为了进一步提高效率，你可以把页表的位置和结构设置得用 **Context** 或者 **XContext** 寄存器就可以直接作为指向页表内部的指针。

既然 MIPS 系统通常在不经地址转换的 kseg0 区域运行操作系统代码，常见的的未命中情形都是用户特权级程序的。为了加异这种常见的情况下常处理，提供了若干硬件特性。首先，这些重填异常被指引到一个不能用于别的异常[3]的

[3]在最初的 MIPS 体系结构上，这是唯一值得拥有自己的入口点的事件。

内存低地址区。其次，用了一系列技巧允许驻留内存的页表被定位在核心的虚拟存储区（ kseg2 区域或其 64 位的对应）这样对于页表当中映射进程地址空间中的“空洞” 的那些部分就不需要物理存储空间了。

最后，**Context** 和 **XContext** 寄存器可以直接从驻留内存的页表访问正确的地址转换数据项。

我们将在第 6.5 节一步步仔细来看这个过程。但是在我们进一步介绍之前，应当注意到使用这些特性并不是强制的。在小型系统上，TLB 可以用作一个从程序（虚拟）地址到物理地址的固定的或很少变化的转换；在这种情况，TLB 甚至都不需要作为高速缓存。

甚至某些为 MIPS 实现的大型虚拟存储的操作系统也没有用“标准”的页表——特别值得一提的是 Linux。这种做法有违 Linux 内核使用存储器的精神，因为 Linux 内核地址映像对所有进程都是一样的。参见第 14.4.8 看看是怎么做的。

6.4.2 使用 ASID

通过对 TLB 表数据项设置一个具体的 ASID，并将 **EntryLo0–1(G)** 位清零，就可以让这些项只在 CPU **EntryHi(ASID)** 寄存器域匹配 TLB 值的时候才用。**EntryHi(ASID)** 是个 8 位的域，允许同时映射最多达 256 个不同的地址空间而不用在进程切换时清除 TLB。如果你确实用光了 ASID，你必须挑选一个可以彻底将其扔出 TLB 的进程。一旦你丢弃了它的所有映射，你就可以收回其 ASID 值并重新用于其它的进程。

6.4.3 Random 寄存器和 Wired 寄存器

硬件并没有给你提供找出哪个 TLB 项最近用得最多的方法。把 TLB 当作高速缓存使用，在需要安装一个新的映射的时候，唯一现实的方法就是随机选择一项进行替换。CPU 通过维护一个随着每个处理器周期递减计数的 **Random** 寄存器来简化实现。

随机替换听起来好象效率会低得可怕；结果可能把最近用得最频繁而且很快又要用到的地址转换数据替换出去。实际上，当有合理数量的可能的牺牲品供选择时，这种现象并不会多到会真的引起问题。

但是，如果能够保证有一些除非明确要求清除否则常驻 TLB 的表项，那么是非常有用的。这在映射已知会经常用到的页时可能有用处，但是真正重要的是在于能让你保证映射的某些页面不会生成重填异常。

稳定的 TLB 项被称为 禁锢*(wired)*的：它们是索引低于编程进 **Wired** 寄存器的值的 TLB 项。TLB 自身并不对这些项作特殊处理，奥妙在于 **Random** 寄存器中，它的值从来不会落入 0 到 wired-1 的范围；它递减计数，但是到了 wired 值的时候就直接回到最大值。所以常规的随机替换保留 TLB 的 0 到 wired-1 项

不变，向那里写入的数据项一直呆到被明确删除为止。

6.5 对硬件友好的页表和重填机制

有一个特殊的地址转换机制，毫无疑问是 MIPS 体系结构设计者心中为 Unix 之类的操作系统中的用户地址保留的。这种方法依赖于为每个地址空间在内存构建一个页表。页表就是用 VPN 做索引的线性数组，其每个数据项的格式和 **EntryLo** 寄存器的位域匹配。TLB 转换后的每对地址需要 2×64 位的数据项，每项 16 字节。

这样减少了在关键的重填异常处理程序上的负担却带来了别的问题。因为每 8 KB 的用户地址空间占用 16 字节的表空间，整个 2G 的用户空间需要 4-MB 的表，这是一块大得令人尴尬的数据块。[4] 当然了大多数用户地址空间只有底部（代码和数据）和顶部（向下增长的堆栈）填有数据，中间有一个巨大的空隙。MIPS 采用的解决方案受到 DEC 的 VAX 体系结构的启发，就是把页表自身也存入核心映射的虚拟存储地址区域（kseg2 或者 xkseg）。这样干净利落地一下子解决了两个问题：

- 节省了物理内存：因为从来不会引用在页表中间未用的空隙，实际上不需要为这些项分配物理内存。
- 提供了一种简单易行的机制，可以在切换上下文的时候重新映射一个新的用户页表，而不用在操作系统中一次找出足够映射所有页表的虚拟地址。取而代之的是，对每一个不同的地址空间有一个不同的内核存储器映射，当你修改 ASID 的值时，指向页表的 kseg2 指针现在自动重新映射到正确的页表。这简直就像是变戏法似的。

MIPS 体系结构支持这种 **Context**（64 位 CPU 中的扩展地址为 **XContext**）形式的线性页表。

如果你让页表在 4-MB 边界的地址开始（因为在虚拟存储器中，任何空隙都不会占用物理存储器空间），并将 **Context** 的 **PTEBase** 域设置为页表起始地址的高位，那么，在用户重填异常之后，**Context** 寄存器就会自动包含重填需要的数据项的地址，不需要做进一步的计算。

到现在一切还好：但是这种方案有导致一个致命的恶性循环的可能：TLB 重填异常处理程序自己可能导致另一个 TLB 重填异常，因为页表的 kseg2 映射并不位于 TLB 中。但是这个问题也可以修正。

如果发生了一个嵌套的 TLB 异常，发生时 CPU 已经处于异常模式了。在 MIPS CPU 中，异常模式中的 TLB 重填被定位到通用异常入口点，在那里被检测并做特殊处理。

[4] 当然这是对于 32 位虚拟地址空间的情形；采用大的地址空间的 64 位 CPU 需要更大的表。

此外，来自异常模式的异常行为也比较奇怪：它并不改变重新开始的地址 EPC，这样当“内层”异常返回时，就直接返回到非异常的 TLB 未命中的位置。看起来好像是硬件开始处理一个异常，然后改变了主意而去处理另一个：但是第二个异常不再是嵌套的，它篡夺了第一个异常。我们看一个例子来了解这是怎样工作的。

6.5.1 TLB 未命中处理

TLB 未命中异常处理总是用一个专用的入口点，除非 CPU 已经在处理一个异常了——就是说，除非 **SR(EXL)** 置位。

下面是在 MIPS32 CPU（或者处理 32 位地址空间转换的 MIPS64 CPU）上处理 TLB 未命中异常的代码：

```
    .set    noreorder
    .set    noat
TLBmiss32:
    mfc0    k1, C0_CONTEXT  # (1)
    lw      k0, 0(k1)       # (2)
    lw      k1, 8(k1)       # (3)
    mtc0    k0, C0_ENTRYLO0 # (4)
    mtc0    k1, C0_ENTRYLO1 # (5)
    ehb                     # (6)
    tlbwr                   # (7)
    eret                    # (8)
    .set    at
    .set    reorder
```

下面是对代码的逐行分析：

(1) k0–1 通用寄存器按照约定是为底层异常处理程序保留的，我们直接使用就好了。

(2–5) 在每一个 TLB 数据项（你也许想要回头看一眼图 6.1 中的 TLB 数据项示意图）中有一对物理方（**EntryLo**）的描述。图 6.4 中所示的 MIPS32/64 Context 寄存器的布局为每一对数据项（每个物理页八个字节的空间）预留了 16 字节，尽管 MIPS32 的 **EntryLo0–1** 是 32 位的寄存器。这是为了与 64 位页表兼容，同时在页表中为保存仅由软件使用的信息提供一些稀疏的域。

在这里交替使用 lw/mtc0 序列能够节省时间：如果你就在紧接的下一条指令使用加载的数据，很少的 MIPS CPU 能够不停顿一直向下走。

如果 kseg2 地址的转换不在页表中，这些加载就容易遭受嵌套的 TLB 未命中异常。我们后面再谈这个问题。

(6) 在从 **EntryLo1** 得到正确的数据之前用 **tlbwr** 写入数据项不是什么好事。MIPS32 体系结构并不能保证紧接该指令后就能准备好数据，但是如果指令之间用一条 **ehb** (execuation hazard barrier) 指令隔开的话，就的确能够保证指令序列的安全性——参见第 3.4 节关于遇险防护的更多信息。

(7) 该行随机替换一个转换对，如上讨论的。

(8) MIP32（还有 MIPS I 之后的所有 MIPS CPU）都有一条 **eret** 指令，从异常返回到 **EPC** 中的地址并且清除 **SR(EXL)**。

那么当你得到另一个 TLB 未命中时会发生什么？来自从异常级的未命中不调用高速的处理程序而是进入通用的异常入口点。我们已经处于异常模式了，所以我们不需要修改异常返回寄存器 **EPC**。

Cause 寄存器和地址异常寄存器（**BadVAddr**、**EntryHi** 甚至还包括 **Context** 和 **XContext**）将与 kseg2 中的页表地址上的 TLB 未命中有关。但是 **EPC** 依然指向引起最初的 TLB 未命中的指令。

异常处理程序将会修理好 kseg2 页表未命中（只要是一个合法的地址），通用异常处理程序将返回到用户程序。当然了，对于原先导致用户空间 TLB 未中的用户地址的转换，我们还什么都没有做，所以立刻又会发生 TLB 未命中。但是第二次发生时，页表转换可用了，用户的未命中处理程序就会顺利完成。漂亮！

6.5.2 XTLB 未命中处理程序

MIPS64 CPU 有两个特殊的入口点。一个——与 MIPS32 共享的——为采用 32 位地址空间的进程进行地址转换；还有一个入口点提供给标记为需要 64 位指针才能用的更大的地址空间的程序。

状态寄存器有三个域，**SR(UX)**、**SR(SX)**、**SR(KX)**，根据地址转换失败时刻的 CPU 特权级选择使用哪个异常处理程序。[5]

当相关的状态位（用户模式的 **SR(UX)** 位）置位时，TLB 未命中异常采用一个不同的向量，那里应该有一个重新加载巨大地址空间转换表的例程。处理程序代码（拥有 64 位地址空间的 CPU 上的 XTLB 未命中处理程序）看上去很象 32 位的版本，除了寄存器为 64 位宽和用 **XContext** 寄存器取代了 **Context**：

```
    .set    noreorder
    .set    noat
TLBmissR4K:
    dmfc0   k1, C0_XCONTEXT
    ld      k0, 0(k1)
```

[5]SR(UX) 位对于用户程序长度加倍让其成为某种“64 位模式”；当为零的时候，用户程序不能使用 64 位整数指令。但是 **SR(SX,KX)** 仅仅用来选择 TLB 重填例程。

```
ld      k1, 8(k1)
dmtc0   k0, C0_ENTRYLO0
dmtc0   k1, C0_ENTRYLO1
ehb
tlbwr
eret
.set    at
.set    reorder
```

注意，结果内核虚拟存储器中的页表结构远远要比这大，毫无疑问将处于巨大的 xkseg 区域。

我应当再次提醒你，这个系统不是强制的，事实上并没有被 Linux 的 MIPS 版本（这是为 MIPS 应用程序设计的进行地址转换的操作系统中最为流行的）所采用。这是 Linux 内核设计中一个相当根深蒂固的设计选择就是内核自己的代码和数据不因进程切换而重新映射，而这一点恰好又是这里讲的 kseg2/xkseg 技巧所必需的。参见第 14.4.8 节介绍的做法。

6.6 MIPS TLB 的日常使用

如果你用的是象 Linux 那样的全功能操作系统，因为系统在你的背后使用 TLB，你自己很少会注意到。如果是一个较为简单的操作系统或者运行时系统，你也许会想还会不会用的到 TLB。因为 MIPS TLB 提供了一个通用的地址转换服务，所以可以有多种方式让你利用它。

TLB 机制允许你从任意映射地址到任意物理地址（以页为单位）进行地址转换，因而可以把程序空间的地址重新定位到你的机器的地址映射中的任意位置。如果你的地址映射需求不大，TLB 能够容纳你需要的所有地址转换，那样就没有必要支持 TLB 重填异常或单独的驻留内存的页表。

TLB 也允许你定义某些地址临时或者永久不可用，这样对这些位置的访问就会导致一个异常来运行操作系统的某些服务例程。通过用普通用户特权级的程序，你可以限制某些软件只能访问你给它指定的地址，通过在转换数据项中使用地址空间 ASID，你可以有效管理多个互相不可访问的用户程序。你也可以对内存的一部分进行写保护。

这一类应用是无穷的，这里给出一个列表来表明各种应用的范围：

- 访问不便于存取的物理地址区域：MIPS 的硬件寄存器位于物理地址范围 0–512 MB 最方便，那里你可以用一个 kseg1 区域的相应指针。但是在硬件无法位于这个区域的地方，你可以把高位物理存储区的任意页面映射到一个方便的映射区域，比如 kseg2。对该 TLB 转换的标志应设置为不用高速缓存进行访问，此后写程序的时候就跟访问方便的位置一样。

- 异常例程的内存资源：假定你想在不用保留寄存器 **k0/k1** 保存上下文的情况下运行一个异常处理程序。如果想这样做，就会遇到麻烦，因为正常情况下，如果不覆盖至少一个寄存器的话，MIPS CPU 就没有地方可以保存任何寄存器。

  你可以用 **zero** 寄存器作为基址，如果用正的偏移量，这些地址就位于 kuseg 区域的前 32 KB，如果是负的偏移量就位于 kseg2 的后 32 KB。没有 TLB 这些地址哪儿都去不了。有了 TLB 就可以把该区域的一个或多个页映射到可读写的存储器，然后利用 **zero** 作基址的存储指令来保存上下文并对你的异常处理程序提供紧急帮助。

- 非-VM 系统中可扩展的堆和栈：即使你没有磁盘或者不想支持完全的请求调页，也能够在有控制的情况下按照需要动态增长应用程序的堆和栈。在这种情况下，你需要 TLB 来映射堆/栈的地址，你要用 TLB 未命中事件来决定是否分配更多的内存或者确定该程序是否失去了控制。

- 仿真硬件：如过有的硬件有时在，有时又不在，那么通过映射的区域访问寄存器可以在正确安装了硬件的系统中直接连到硬件，在其它情况下调用软件处理程序。

总的指导思想就在于，TLB 因为其要适合大型操作系统的规范而具有一些精巧的设计，这些设计对于程序员而言就成了一个可以直接使用的通用资源。

6.7 更简单的操作系统中的内存管理

一个为桌面以外的系统设计的操作系统通常叫做实时操作系统（RTOS），借用了一个本来指与实时有关的东西的术语。[6] 本章第一部分列出的类似 Unix 的系统拥有更小的操作系统可能具备的一切东西，但是许多 RTOS 则要简单得多。

你可能碰到的一些操作系统产品包括风河公司的 VxWorks、Express Logic 的 Thread/X 以及 Mentor（在收购了 Accelerated Technology 之后）的 Nucleus。这几个都提供在单个地址空间运行多线程的能力。没有任务之间的保护——假定运行在上面的软件只是单个紧密集成的应用程序。在许多情形中，OS 运行的时间真的很少，供应商的主要精力放在了给开发者提供构建、调试和剖析工具上。

对许多不同种类的嵌入式系统，使用一个象 Linux 这样更为复杂的操作系统是否值得，目前尚未形成一致看法。好处是你将得到一个更为丰富的编程环境、在集成系统时极有价值的任务到任务间的保护，以及可能更加清晰的接口。值得花费额外的存储器和 CPU 计算能力，同时损失一些对时间的控制，以换取

[6] 真正的“实时”涉及到与时间期限有关的编程，非常有意思也很有难度，但是并非主流的追求。

一个更加智能的系统吗？电视机顶盒、DVD 播放机、家庭网络路由器制造商觉得使用 Linux 值得：另一些系统（复杂度不见得有多大的差别）依然习惯于使用简单的操作系统。

当然了 Linux 是开放源码的。有时候没有授权费本身就是优势；也许更重要的是，如果由于操作系统的缺陷而让你的系统无法运作，开放源码意味着你可以自己修正缺陷，或者花钱请专家帮助你解决——马上就可以做到。下面这句话听起来好象有些自相矛盾：一个商业的操作系统越成功，要在合理期限内找人修复缺陷反而越困难。

但是就目前来讲，作为一个开发者，几乎任何情况都有可能要面对。当你试图搞懂一个新的存储器管理系统时，首先要做的是，搞明白存储器映射，包括呈现给应用软件的虚拟映射和呈现给系统的物理映射。正是采用了单纯的虚拟地址映射才使得 Unix 的存储器管理描述起来相对简单。但是定位于嵌入式应用的操作系统通常起源于没有存储器管理功能的硬件，其进程的存储器映射常常夹杂有一些没有映射的地址隐藏在其中。用铅笔、纸张还有耐心就不难搞明白。

第 7 章　浮点支持

1987 年，MIPS FPU 为买得起的工作站上的微处理器数学性能设立了新的里程碑。与 CPU 基本上只是简单直接的实现主要依赖于基本的体系结构来获得性能不同，FPU 是一个充满了创新和智慧的传奇式的硅半导体设计。

后来，MIPS FPU 受到 SGI 对一度只属于超级计算机的保留地的数学性能的需要而一路提升。浮点计算在嵌入式系统中的使用增长得相当缓慢；但是对于浮点需求的总趋势是在硬件复杂性与软件简单性和易维护性之间进行权衡，这个经典的权衡结果（随着时间）只会朝一个方向发展。有一天，所有“主流应用”的处理器都会离不开浮点功能。

7.1　有关浮点数的基本概念

浮点数学带有许多神秘性。你也许对它的概念非常清楚，但对细节可能就模糊了。本节介绍一下浮点数据的各个组成部分以及各自代表的意义。这样做我们可能讲了很多你已经知道的知识；请跳过本节但是要留意课文。

处理非常大或者非常小的数的人习惯于采用指数/科学记数法；例如，从地球到太阳的距离为

$$93 \times 10^6 \text{ 英里}$$

这个树被定义为一个*尾数(mantissa)* 93 和一个*指数(exponent)* 6。

同样的距离也可以写为：

$$9.3 \times 10^7 \text{ 英里}$$

数值分析人员更喜欢用第二种形式；一个十进制的指数和一个介于 1.0 和 9.999... 之间的尾数，称为规格化。[1] 规格化的形式对于计算机表示更有用，因为我们不需要额外的信息记录小数点的位置。

计算机保存的浮点数是指数形式的，但是以 2 为底而不是以 10 为底。尾数和指数都是二进制的。仅仅把指数部分换成 2 的幂，上面引用的距离可以写为：

$$1.38580799102783203125 \times 2^{26} \text{ 英里}$$

[1] 在这种形式中，尾数也叫做“小数部分”或者“小数” ——当然这样更容易记忆。

尾数可以表达为一个二进制小数，就跟十进制一样；比如：

$$1.38580799102783203125 = 1 + 3 \times \frac{1}{10} + 8 \times \frac{1}{100} + 5 \times \frac{1}{1000} + \ldots$$

就等于二进制的值：

$$1.01100010100010001000101 = 1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{4} + 1 \times \frac{1}{8} + \ldots$$

但是尾数和指数都不是象这样以标准形式存储的——要理解为什么这样，有必要先回顾一下历史。

7.2 IEEE 754 标准及其历史背景

因为浮点处理的是数的近似表示（就像小数一样），过去的计算机实现对于处理非常小或很大的数的行为时在细节上各不相同。这就意味着，数值计算例程，即使代码完全一样，其行为在不同计算机上可能有差别。在某种意义下，这些差别不应该有什么问题：在没有哪个实现能够产生一个真正“正确”的答案的情形下，只能得到不同的答案。

使用计算器的显出由此而导致的这种恼人的现象：如果计算一个整数的平方根，然后再平方，很少能够得到原来的整数，而是带有很多 9 的结果。

数值计算例程本质上难以编写，而且难以证明其正确性。很多大量使用的函数（比如常见的三角函数）就是通过重复的逼近计算的。这样的例程可能在一种 CPU 上可靠地收敛到正确的结果，但是换一个 CPU 对某个难算的值可能就会陷入死循环。

ANSI/IEEE 754 标准——1985 IEEE Standard for Binary Floating Point Arithmetic（通常简称为 IEEE 754）——出台就是为了解决这种混乱状况的。该标准精确定义了一小类基本操作应当产生的结果，包括在极端条件下的结果，保证程序员不管用什么机器，只要是同样的输入就能够得到同样的结果。所用的方法就是从每一种支持的数据格式中获取尽可能高的精度。

也许 IEEE 754 的选项太多了，但相对于此前的混沌状态是一个巨大的进步；自从 1985 年成为真正的国际标准以来，IEEE 754 已经成了所有新实现的基础。

IEEE 754 规定的操作包括 MIPS CPU 硬件可以执行的全部浮点操作，还要加上一些必须由软件仿真的操作。IEEE 754 规定了下列的操作：

- 结果的舍入和精度：即使简单运算的结果也有可能无法用有限小数表示；例如十进制数：

  $$\frac{1}{3} = 0.33333\ldots$$

  即无限循环，无法精确写出。IEEE 754 允许用户有四种选项：向上舍入、向下舍入、向零舍入或者最近舍入。舍入的结果就是相当于以无限精度计

算之后再舍入。当无限精度的结果恰好介于两个可表示的形式中间时，最近舍入会导致二义性；按照规定这种情况应该选择最低有效位为零的值。

- 什么时候一个结果被看作是异常的？IEEE 754 自己定义了异常这个词。可以产生异常的计算结果包括：
  - 无意义，比如求 -1 的平方根（“无效(invalid)”）
  - “无限”，直接或间接除以零导致的结果
  - 太大而无法表示（“上溢”）
  - 太小而导致表示会丧失精度（“下溢”）
  - 无法精确表示，比如$\frac{1}{3}$ （“不精确”）——不必说，对于大多数目的来说，都可以接受最接近的近似值

  所有这些合在一起，统称为异常的结果。

- 当一个操作产生异常结果时所采取的动作：对于上面列出的每个异常结果类，用户可以在以下几种方式中选择：
  - 用户可以让计算中断，让程序以某种依赖于操作系统或者编程语言的方式发出信号。部分是由于该标准并没有对用户异常定义一种到具体语言的绑定，这种方式几乎从来没有用过。一些 FORTRAN 编译系统对这种情况产生致命错或无穷大的结果。
  - 最为常见的是，用户程序并不想知道 IEEE 754 异常。在这种情况下，该标准规定应该产生哪个值。上溢和除以零产生无穷大（分为正负无穷大）；无效的操作产生两种风格的非数 NaN (not a number)，分别称为“默许（quiet）”式和“明令（signaling）”式。取值很小的数产生一个“非规格化”的表示，精度逐渐降低直至消失为零。

    该标准同时定义了对异常值进行操作的结果。无穷大和 NaN 一定产生进一步的无穷大和 NaN，但是一个默许式 NaN 作为操作数不会触发异常报告机制，一个明令式 NaN 用作计算时会导致新的异常。

  大多数程序不用 IEEE 754 的异常报告，而是依赖于系统产生正确的异常值。

7.3 IEEE 浮点数的存储方式

IEEE 推荐了在几个不同大小上编码浮点数的不同二进制格式。但都有一些共同的精巧的特性，这些特性是基于此前混沌的年代里实现人员的经验之上的。[2]

[2]IEEE 754 是制定良好标准的过程的一个典范；相当长时间的一段混沌的实验期可以让人们找出好的做法进一步发展，然后由一些对于技术有深刻理解并且意志坚定的用户（此处即数值计算的

第一件事是指数并没有以有符号的二进制数的形式保存，而是采用偏置*(biased)* 方式，这样一来指数域永远为正：指数值为 1 代表最小的（负数）合法的指数值；对于 64 位 IEEE 格式，指数域为 11 位的长度，可以容纳从 0 到 2047 的值。指数值 0 和 2047（全 1 的值）保留用于特殊目的（我们呆会儿会讲到），所以我们可以表示的指数范围就是从 -1022 到 1023。

对于数：

$$\text{尾数} \times 2^{\text{指数}}$$

我们在指数域实际存储的值是

$$\text{指数} + 1023$$

偏置指数（还有仔细安排各个域的次序）会产生有用的效果，保证浮点比较（相等、大于、小于等）和两个由同样的位构成的带符号的整数比较得到的结果相同。随之浮点比较就可以通过廉价、快速和熟悉的逻辑提供。

7.3.1 IEEE 尾数和规格化

IEEE 格式使用一个和尾数分开的符号位（0 代表正号，1 代表负号）。所以存储的尾数只要表示整数就行了。所有 IEEE 格式适当表示的数都是规格化的，即：

$$1 \leq \text{尾数} < 2$$

这意味着尾数的最高有效位（在小数点前的那一位）永远为 1，所以我们实际上不需要存储它。IEEE 标准称之为 隐藏位。

所以现在数 93 000 000 规格化的表示的二进制尾数为1.01100010110001000101，二进制指数为 26，用 IEEE 64 位格式表示设置这些域：

$$\text{尾数域} = 0110001011000100010100 0\ldots$$

$$\text{指数域} = 1049 = 10000011001$$

从另外一个角度看来，带有指数域和 E 和尾数域 m 的 64 位 IEEE 数表示的值 f, 其中：

$$f = 1.m \times 2^{E-1023}$$

（只要接受 1.m 代表二进制的小数，小数点前面为 1 ，后面是尾数域的内容）。

编程人员）构成的小型委员会加以标准化。然而，标准委员会的整个生态环境，也许研究起来很有意思，却偏离了这一典范。

7.3.2 为特殊值使用而保留的指数值

指数域最小和最大的值用来表示非法的数量。

$E == 0$ 用来代表零（尾数也为零）和由于太小而不能用标准形式表示而采用非规格化的形式表示的数。E 为零、尾数为 m 的非规格化数表示 f，其中：

$$f = 0.m \times 2^{-1022}$$

随着非规格化的数变小，精度会逐渐丧失。目前为止生产的 MIPS CPU 不能够处理非规格化的运算结果或操作数，结果或操作数为非规格化的运算就会直接被投射到软件异常处理程序。现代的 MIPS FPU 可以配置成用零代替非规格化的结果继续运算。

$E == 111\ldots1$（即 2047 在 IEEE 双精度浮点数采用的 11 位域中的二进制表示）用来代表下列：

- 尾数为零时，代表非法值正负无穷大（用通常的符号位区分）。
- 尾数不为零时，代表明令非数 NaN。对于 MIPS，尾数的最高有效位确定 NaN 是否为默许(quiet)非数 (0) 或者明令(signaling)非数 (1)。这个选择不属于 IEEE 标准的一部分，而且和大多数其它兼容 IEEE 标准的体系结构所用的约定相反。[3]

7.3.3 MIPS FP 数据格式

MIPS 体系结构采用了 IEEE 754 推荐的两种浮点格式：

- 单精度：可以放进 32 位的存储空间。MIPS 的编译器用单精度表示 float 类型。
- 双精度：要用 64 位存储空间。C 语言编译器用双精度表示 C 的 double 类型。

存储器和寄存器的布局如图 7.1 所示，还配有一些例子演示数据怎样排列。注意到 float 类型的表示无法精确地容纳象 93 000 000 这样的数。

构成一个双精度数的两个字在内存中的顺序（最高有效位在先，或者最低有效位在先）取决于 CPU 的尾端——在第 10.2 节的末尾有讨论。这个顺序总是和整数单元的尾端保持一致。

下面的 C 结构体定义了 MIPS CPU（大多数 MIPS 工具链是这样的，但要注意，一般来说，C 结构体的布局取决于具体的编译器而不仅是目标 CPU）上的两种浮点类型的域：

[3]我相信只有 HP Precision 作了和 MIPS 同样的选择，这点暗示二者之间有某种传承关系。任意的 1/0 选择是计算机技术中的线粒体 DNA。

```
#if BYTE_ORDER == BIG_ENDIAN

struct ieee754dp_konst {
    unsigned sign:1
    unsigned bexp:11;
    unsigned manthi:20; /* cannot get 52 bits into ...  */
    unsigned mantlo:32; /* ... a regular C bitfield     */
};

struct ieee754sp_konst {
    unsigned sign:1
    unsigned bexp:8;
    unsigned mant:23;
};

#else  /* little-endian */

struct ieee754dp_konst {
    unsigned mantlo:32;
    unsigned manthi:20;
    unsigned bexp:11;
    unsigned sign:1
};

struct ieee754sp_konst {
    unsigned mant:23;
    unsigned bexp:8;
    unsigned sign:1
};

#endif
```

7.4 MIPS 对 IEEE 754 的实现

IEEE 754 的要求很严格，这导致了两个问题。首先是内建异常结果检测能力的要求使得流水线很难实现。你也许想用这种做法来实现 IEEE 异常信令机制，但是更深层次的原因是能够检测某些硬件无法产生正确结果而需要帮助的

Single

	31	30 23	22 0
	Sign	Exponent	Mantissa
93000000	0	0001 1010	101 1000 1011 0001 0001
0	0	0000 0000	000 0000 0000 0000 0000
+Infinity	0	1111 1111	000 0000 0000 0000 0000
-Infinity	1	1111 1111	000 0000 0000 0000 0000
Quiet NaN	x	1111 1111	0xx xxxx xxxx xxxx xxxx
Signaling NaN	x	1111 1111	0xx xxxx xxxx xxxx xxxx

Double

	31	30 20	19 0 31 0
	Sign	Exponent	Mantissa
93000000	0	000 0001 1010	1011 0001 0110 0010 0010 1000 0000 ....
0	0	000 0000 0000	0000 0000 0000 0000 0000 0000 ....
+Infinity	0	111 1111 1111	0000 0000 0000 0000 0000 0000 ....
-Infinity	1	111 1111 1111	0000 0000 0000 0000 0000 0000 ....
Quiet NaN	x	111 1111 1111	0xxx xxxx xxxx xxxx xxxx xxxx ....
Signaling NaN	x	111 1111 1111	xxxx xxxx xxxx xxxx xxxx ....

图 7.1: 浮点数据格式

情况。

如果用户选择当 IEEE 异常结果产生时被告知，那么要想这点有用的话，异常发生就应该是同步的；[4] 自陷发生后，用户将看到所有前面的指令都执行完毕，而且所有的浮点寄存器仍然处于指令执行前的状态，还要保证后续指令还没有产生任何效果。

在 MIPS 体系结构中，硬件自陷（如 5.1 节所示）传统上就是这样的。这一点的确限制了浮点操作流水化的机会，因为你要直到硬件确保浮点操作不会产生自陷的时候才能提交后面的指令。为了避免增加执行时间，浮点操作必须在读取操作数后的第一个时钟周期阶段决定是否自陷。对于大多数异常结果，FPU 可以可靠的猜测并且为任何可能自陷的计算[5]停止流水线；然而如果你将 FPU 配置为采用硬件信号报告 IEEE 非精确异常结果，那么所有的浮点流水线操作都被禁止，一切都会慢下来。你可能不想那样做。

关于 IEEE 754 的第二个大问题就是对异常结果的使用，特别是对于非规格化数据——属于合法操作数。象 MIPS FPU 这样的芯片设计逻辑单元是高度结构化的，异常结果不太容易放下。在正确的结果超出硬件的表示能力的地方，就会发生自陷并在 **Cause(ExcCode)** 域放入一个未实现的操作码。这样直接的后果就是异常处理例程对浮点应用变得必不可少。现代的 MIPS CPU 常常在 **FCSR** 控制/状态寄存器中包含一个或多个与实现相关的选项位，当你准备以牺

[4]在本书的其它部分和 MIPS 的文档中，你将会看到这个条件被称作“精确异常”。但是鉴于在 IEEE 标准中“精确”和“异常”都已经被用作了另外的意义，我们在这里就称之为“同步自陷”。抱歉这里的混淆。

[5]有些 CPU 对此采用启发式方法，有时候会因为一个最后并未自陷的操作而停止流水线；这只是一个性能问题，如果不是经常发生问题就不大。

牲严格的 IEEE 754 兼容性为代价换取性能时可以设置这些位——可能性更大的是——为了避免个别情形下的“未实现的操作”自陷而设置这些位。

7.4.1 所有 MIPS CPU 都需要浮点自陷处理程序和仿真程序

MIPS 体系结构并没有确切描述哪些计算的执行无需软件干预。对于任何真正实用的浮点代码，完整的软件浮点仿真程序都是必要的。

在实践中，FPU 只在程序可能产生的计算的很小一部分上才会自陷。简单的使用浮点极有可能从来都不会产生硬件处理不了的情形。

一个看上去包含了适当情形的很好的经验法则如下：

- MIPS FPU 当一个操作应当产生除了非精确和溢出之外的任何 IEEE 异常和异常结果时都发生未实现的自陷。对于溢出，硬件会产生一个无穷大或最大可能的值（取决于当前的舍入模式）。FPU 硬件不会接受或者产生非规格化的数以及 NaN。
- MIPS FPU（除了最早期的以外）给你提供一个用于下溢的非 IEEE 的可选模式，对一个非规格化的（微小的）结果可以自动写成零。

未实现的自陷是 MIPS 体系结构实现的一个技巧，和 IEEE 的异常的标准条件有很大不同。你可以运行一个程序并忽略 IEEE 异常，而受影响的指令将会产生一个明确定义的异常值；但是你不能忽略未实现的自陷，而不产生无意义的结果。

7.5 浮点寄存器

MIPS CPU 拥有 32 个浮点寄存器，通常记为 **$f0–$f31**。除了一些真的很老的（MIPS I）CPU 之外，每个浮点寄存器都是 64 位，能够容纳一个双精度的值。

最早的 MIPS CPU 只有 16 个浮点寄存器。在某种意义上也可以说是有 32 个 32 位的寄存器，但是每个奇/偶编号的对组成一个数学单元（当然包括双精度浮点数学了）。奇数编号的寄存器仅在执行加载、存储和在整数浮点寄存器之间传送数据的时候才用到。[6] 如果你告诉汇编器你在为老的 CPU 生成代码，就会通过一对 load/store 机器指令来合成双倍宽度的传送；当写 MIPS I 兼容代码的时候，你从来不需要看到奇数编号的寄存器。

MIPS I 消失已经很久了，但是后来的 CPU 在 **SR(FR)** 中加进了一个“兼容位”——保持为 0 就会得到 MIPS I 兼容操作。仍然有不少软件以这种方式工作。看起来使用更多的浮点寄存器好像是没有头脑——但是这不是个人可以选

[6] 值得强调的一点是：奇数编号寄存器的使用不受 CPU 的尾端影响。

择的；你需要检查你的编译器支持什么，整个系统（包括所有的库和其它引入的代码）需要和编译器的寄存器用法保持一致。

还值得指出的一点就是 MIPS FP 寄存器有时用来存储和操作有符号的整型数据（32 位或者 64 位）。特别是，当程序进行整数和浮点数据转换时，这些转换操作完全在 FPU 中进行——在浮点寄存器中把整数数据转换成为浮点数据。

7.5.1 浮点寄存器的传统习惯命名和用法

表 7.1: 浮点寄存器使用约定

<table>
<tr><th></th><th colspan="3">ABI</th></tr>
<tr><th></th><th>o32</th><th>n32</th><th>n64</th></tr>
<tr><td>函数返回值</td><td colspan="3">$f0、$f2</td></tr>
<tr><td>参数寄存器</td><td>$f12、$f14</td><td colspan="2">$f12–$f19</td></tr>
<tr><td>函数调用间要保存（适用于寄存器变量）</td><td colspan="2">偶数$f20–$f30</td><td>$f24–$f31</td></tr>
<tr><td>临时变量（函数调用之间不保存或“由调用者保存”）</td><td>偶数$f4–$f10、$f16、$f18</td><td>偶数 $f4–$f10、$f16、$f18、所有奇数 $f1–$f31</td><td>$f1、$f3–$f11、$f20–$f23</td></tr>
</table>

跟通用寄存器一样，MIPS 调用约定加上了一整套与硬件无关的寄存器使用规则；告诉你哪些浮点寄存器用来传递参数，哪些寄存器的值在函数调用之间要保存等等。表 7.1 给出了三种最常见的 ABI。应用程序二进制接口 ABI(Application Binary Interface) 是有关约定的综合陈述，这些约定允许模块——可能是用不同的工具编译好的——能够成功的粘合到一起成为一个程序，并能在符合要求的操作系统上运行。在 11.2 节我们进一步讨论 API；就目前而言，只要说 o32 约定用于老式 MIPS I CPU 的 16 个寄存器的安排，n32 和 n64 约定（不要管前一个名字中的 32）只能用于 64 位 CPU。

功能的分配基本上和整数寄存器一样，没有特殊情形。但是由于历史上不存在奇数编号寄存器的原因，浮点寄存器功能分配要乱不少。

7.6 浮点异常/中断

当一个浮点操作不能产生正确的结果时，或者被要求在某些或者全部 IEEE 异常结果上发生自陷时，就会发生一个 MIPS 异常，如第 5 章所述，Cause 寄存器描述如第 3.3.2 节讨论的。

浮点异常总是精确的：在你到达异常处理程序的那一点，导致异常的指令和任意后续指令看上去就象从未发生过。EPC 将会指向重新开始指令的正确地

表 7.2: 浮点控制寄存器概述

约定名称	CP1 控制寄存器编号	描述
FCSR	31	详尽的控制寄存器——历史上的 MIPS CPU 中唯一的 FPU 控制寄存器，包含了所有的控制位。但在实际中，有些位通过 **FCCR**、**FEXR** 和 **FENR** 存取更方便。参见下文。
FIR	0	FP 实现寄存器：关于本 FPU 能力的只读信息，在下一节讲述。
FCCR **FEXR** **FENR**	25 26 28	**FCSR** 部分信息的重新组织，允许在不影响无关的位的情况下更新域。但是在不完全兼容 MIPS32/64 的 CPU 上很可能没有。**FCCR** 有一个浮点条件码，**FEXR** 含有你读取的 IEEE 异常条件信息（原因和标志位），**FEXR** 有（可写的）IEEE 异常条件使能域。

方。如第五章所述，EPC 要么指向发生异常的指令，要么指向紧挨着的前一条分支指令。如果是分支指令，状态寄存器 SR(BD) 就会置位。

7.7 浮点控制：控制/状态寄存器

关于 FPU 硬件的信息和改变可选行为的控制信息以协处理器控制寄存器的形式提供，通过 **ctc1** 和 **cfc1** 指令来访问，这两条指令分别将数据传送到或者传送出控制寄存器（当然是和通用寄存器之间传送了）。

其中最常见的就是浮点控制/状态寄存器 **FCSR**，[7] 把用户关于浮点操作的选项的信息和控制域合在了一起。关于硬件厂商和能力的信息存放在一节讲的实现/版本寄存器 **FCR0** 中。

使用 **FCSR** 曾经成为一场恶梦，因为其把读写域混合在一起了：所以在 MIPS32/64 CPU 中有三个辅助的寄存器 **FCCR**、**FEXR** 和 **FENR** 以更容易处理的方式提供同样的功能。

这样就形成了好几个寄存器，请参见表 7.2。

下面是对图 7.2 的一些说明。标记为 0 的域读出为 0，写入时应当也为 0。

- FCC7–1, FCC0: 这些是条件位，由浮点比较指令设置、条件分支指令测试。最先有的 FCC0，以前曾经称为 FCSR(C) ——但是所有现代的 MIPS CPU 都提供 8 位。FCCR 寄存器——如果你的 CPU 提供的话——给你提供了把所有条件位集中到一起的服务。

  注意在这里，还有其它地方，浮点实现违背了我们在第 1 章讲过的 RISC 原则。这有好几个方面的原因：

[7] 以前 **FCSR** 曾被称为 **FCR31**，但是我更愿意用容易记忆的符号。

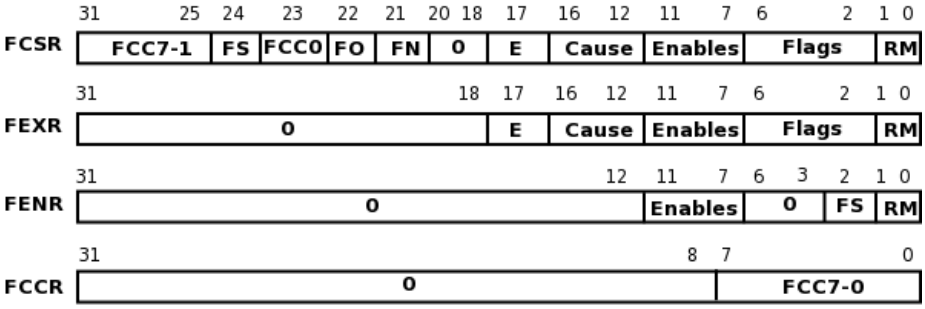

图 7.2: FPU 控制/状态寄存器域

- 最初的 FPU 是一个单独的芯片。测试浮点条件的条件分支不得不在整数单元（整数单元要负责找出分支的目标）内部执行，这样就离浮点寄存器很远。单个的条件位对应于单个的硬件信号。
- 浮点运算太复杂，无法在一个时钟周期内执行完毕，所以对整数很有效的简单流水线对浮点不能提供最佳性能。

分支和测试指令都有一个额外的三位的域可以选择指定八个可能的条件位中的一个来设置或者测试。在 MIPS III 或者更早期的 CPU 上，只有一个条件位，这种位域都为零。额外的条件位对于软件流水线的条件浮点代码具有重要价值；参见第 8.5.7 节。

- FS/FO/FN: 这些设置影响 CPU 在面临太小而无法用标准浮点表示的数的时候的行为。如果你置位任何一个，你的系统就不再符合 IEEE 754 标准，但可以避免一些“未实现的”自陷。

  如果你的 CPU 是符合 MIPS32/64 的，这些域就都有。如果不是，那么会实现 FS 域，至于其它的域，就得查对 CPU 手册了。

  FS (flush to zero) 导致一个太小而无法采用标准表示的结果（非规格化的结果）被悄悄用零代替。因为已知的 MIPS CPU 硬件在这种情况下都不能生成正确的结果，这种做法就可以避免（缓慢的）到浮点仿真软件的自陷。

  FO (flush override) 和 **FS** 联合起来检测非规格化的操作并在内部用零代替。如果输入是非规格化的，单独设置 **FS** 不会避免自陷。

  FN (flush to nearest) 和 **FS** 联合使用，通过强制非规格化的数为最接近的规格化的值（有时是可以表示的最小的数，而不一定是零）而改善精度。

- E: 在 FPU 自陷之后，该位被置位以标志未实现的指令异常。[8]

  每当 FPU 不能执行指令（但是指令的确是协处理器 1 的编码），或者当 FPU 无法确定在这些操作上用这些操作数是否能产生符合 IEEE 754 标准

[8] MIPS 的文档看上去略有不同，因为把这位当作 **Cause** 域的一部分。

的正确结果或异常信令时，该位就被置位同时发生中断。

不管什么原因，当 **E** 位置位时，你应当安排相应的指令由软件仿真程序重新执行。

任何 FPU 操作发生“未实现的”异常时，目标寄存器的内容都保持不变，而浮点 **Cause** 域为不确定。

- Cause/Enables/Flags: 每个都是 5 位的域，每个 IEEE 异常类型对应于一位：

V	位 4	无效操作（比如 -1 求平方根）
Z	位 3	除以零
O	位 2	上溢，结果太大而无法表示
U	位 1	下溢，结果太小而无法表示
I	位 0	非精确（很少用——即使 1 除以 3 用二进制也不能精确表示）

  三个不同域的工作如下：

  - Cause: （由硬件或者仿真软件）根据最近完成的浮点指令的结果设置。从 FEXR 中读取最容易。
  - Enables: 当操作产生一个异常结果可能会设置相应的 **Cause** 位时，如果其中一位置位，那么 CPU 就会发生自陷，这样软件可以采取必要的措施报告异常结果。你既可以通过 **FSCR** 也可以通过 **FENR** 来设置这些位。
  - Flags: 这些位是 **Cause** 位的“顽固”版本，也是自从上次寄存器清零以来发生的异常结果的逻辑“或”。**Flags** 位只能通过写入 **FCSR** 或 **FEXR** 来再次清零。

- RM (rounding mode): 这是 IEEE 754 规定必须实现的。允许的值如表 7.3 所示。

  许多系统定义最近舍入 RN 为默认的行为。你可能从来不会用到别的舍入方式。

体系结构保证如果一个操作不设置 **FCSR(E)** 位但是却设置了一个 **Cause** 位，那么 **Cause** 位的设置和产生的结果（如果相应的 **Enable** 位关闭）就都符合 IEEE 754 标准。

MIPS FPU 依赖软件仿真（即采用未实现的自陷）来达到几个目的：

- 任何含有非规格化的操作数或者结果下溢（产生非规格化的结果）的操作将会自陷到仿真程序。仿真程序自己必须测试下溢使能位是否置位，要么触发 IEEE 异常，要么生成正确结果。

表 7.3: 浮点控制寄存器中的舍入模式编码

RM 值	描述
0	RN (Rounding to nearest 最近舍入)：将结果舍入到最接近的可表示的值；如果结果恰好位于两个可表示的值中间，舍入到零。
1	RZ (Rounding to zero 向零舍入)：将结果舍入到其绝对值小于或等于无限精确值的最接近的可表示的值。
2	RP (Rounding up, or toward+infinity 向上舍入，或向正无穷舍入)：将结果向上舍入到下一个可表示的值。
3	RN (Rounding down, or toward-infinity 向下舍入，或向负无穷舍入)：将结果向下舍入到下一个可表示的值。

- 应当能够正确识别产生无效自陷的操作，这样如果使能 IEEE 异常，仿真程序就什么都不用做。但是如果禁止 IEEE 无效异常，就会调用软件仿真程序，因为硬件不能生成适当的结果（通常是默许 NaN）。

  对于操作数为明令 NaN 的情况处理完全相同。

- 大多数浮点硬件对于常规算术运算可以处理上溢（取决于舍入模式，要么生成有限的极大值，要么是正负无穷大）。但是需要软件仿真程序实现一个发生溢出的浮点到整数的转换操作。

Cause 位在未实现的操作自陷到仿真程序后为未定义的。

通常的做法是提供一个全功能的仿真程序（能够在没有浮点的 CPU 上提供 IEEE 兼容的算术运算）作为 FPU 硬件的后备。如果你的系统提供的比这还少，就很难找出哪里才可以安全省略功能。

7.8 浮点实现寄存器

检查 **Config1(FP)** 之后查看只读的 **FIR** 寄存器以找出你是否真的有浮点单元：**FIR** 的域告诉你它能做什么，以及（偶尔）它是什么版本——参看图 7.3。

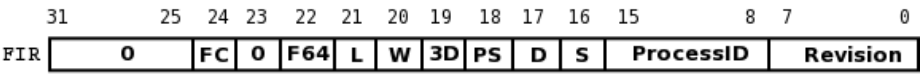

图 7.3: FPU 实现/版本寄存器

这些域如下：

- FC(full convert range)：硬件会完成任何转换操作（浮点类型之间以及浮点和整数类型之间）而不会因用完所有位导致“未实现”异常。

- F64: 这表明有 32 个全双精度的浮点寄存器——就是说，不是老式的 MIPS I 风格的浮点单元。
- L/W/D/S: 单个的位标志该 CPU 是否实现 64 位整数（L），32 位整数（W），64 位浮点双精度（D），32 位浮点单精度（S）操作。如果不能全部支持这些，那你需要一些非同寻常的软件支持。
- 3D/PS: 表明是否支持两个可选指令集。PS 意味着拥有所有算术操作的单精度对 SIMD 版本，以 .ps 后缀为标志：每个指令在装入同一个寄存器的一对单精度值上同时进行两个操作。参见第 7.10 节。

  3D 表示支持 MIPS-3D 指令集扩展。这是一个很小的扩展，只有在支持单精度对之后才有用，所以也放在第 7.10 节讲。
- ProcessorID：典型情况下，返回与整数单元在协处理器 0 的 **PRId** 寄存器中的处理器 ID 相同的值。也许更有用的是，如果真的有 FPU 硬件可用的话，读取时应该为非零值。在你读取该域之前或者成功运行任何浮点运算之前，你或你的操作系统需要将 CP0 的“状态”寄存器中的“协处理器 1”使能位 **SR(CU1)** 切换到允许状态。
- Revision: 该域取决于实现者；保持该域对测试提交工程师可见可能有用，当需要用软件实现克服某些硬件错误的时候可能希望测试该域。除此之外，软件应当坚决忽略这个域的值。

7.9 浮点指令指南

本节按照功能分类讲解所有标准的浮点指令。下一节讲“单精度对”和 MIPS-3D 扩展提供的可选指令。

浮点指令按照助记符字母次序列在表 8.4 中，二进制编码和其它 MIPS 指令一起列在表 8.6 中。

我们把这些指令分成下列几类：

- Load/store: 在浮点寄存器和存储器之间直接传送数据。
- 寄存器间传送: 在浮点寄存器和通用寄存器之间传送数据。
- 三操作数算术指令：一般的加减乘除等等。
- 乘加操作：新奇（明显非 RISC）的高性能指令，MIPS III 及之前没有。
- 改变符号：简单的操作，分开来讲是因为其简单的实现永远不会导致 IEEE 异常。
- 转换操作：单精度、双精度和整数值之间的转换。
- 条件分支和测试指令：当浮点单元与整数流水线再次碰面的地方。

7.9.1 Load/Store

这些操作将 32 位或 64 位数据从存储器加载到浮点寄存器，或者从浮点寄存器存储到存储器。在加载和存储时，要注意以下几点：

- 不转换也不察看数据，即使表示的是个无效的浮点数值也不会引发异常。
- MIPS I 风格的 32 位 FPU 只能在偶数编号的寄存器上进行数学运算。对于奇数编号寄存器的加载和存储可以让你访问 64 位值的另一半。
- MIPS I 和 MIPS II CPU 允许在浮点加载数据生效之前花费一个额外周期；不允许对那个数据的使用执行互锁（同样的规则也适用于对通用寄存器的加载）。编译器/汇编器通常会为你处理这些，但是在老式的 CPU 上浮点加载随后紧跟一条使用加载值的指令是无效的。
- 在写汇编程序时，“合成”形式的指令（比如 **l.d** ）应该比本地的机器指令（**ldc1**）优先使用；“合成”形式易读而且可用于所有的 CPU。汇编可以用多条指令来实现 CPU 未实现的机器指令。一个合成的 load/store 操作可以用汇编器支持的任何寻址方式（如第 9.4 节所述）。
- 浮点加载/存储操作的地址必须对齐到要加载的目标大小边界——单精度或者字对齐到四字节边界，双精度和 64 位整数类型对齐到八字节边界。

在机器指令描述中，(`disp` 为有符号的 16 位量，“*”为 C 语言的取指针内容操作符)：

```
lwc1 fd, disp(s)      fd = *(s + disp);
swc1 fs, disp(s)      *(s + disp) = fs;
```

我所知道的 MIPS64/MIPS32 兼容的 FPU 都支持‘64 位 load/store:

```
lwc1 fd, disp(s)      fd = (double) *(s + disp);
swc1 fs, disp(s)      *(s + disp) = (double) fs;
```

MIPS32/64 增加了双寄存器的索引寻址：

```
lwxc1 fd, i(s)      fd = *(s + i);
swxc1 fs, i(s)      *(s + i) = fs;
ldxc1 fd, i(s)      fd = (double)*(s + i);
sdxc1 fs, i(s)      *(s + i) = (double)fs;
```

但是事实上，当你写汇编的时候，这些都不需要记住。“addr”可以是汇编器支持的任意寻址方式：

```
l.d fd, addr        fd = (double) *addr;
l.s fd, addr        fd = (float) *addr;
s.d fs, addr        (double) *addr = fs;
s.s fs, addr        (float) *addr = fs;
```

汇编器将生成适当的指令，允许从几个有效的寻址模式中进行选择。在 32 位 CPU 上双精度数加载将会汇编成两条机器加载指令。

7.9.2 寄存器间传送

当数据在整数和浮点寄存器之间传送时，不做任何数据转换，也不会发生任何异常。但是当数据在浮点寄存器之间传送时，你需要指定具体的浮点数据类型，企图传送一个在指定类型中无意义的值——虽说不会产生异常——不一定会正确拷贝所有的位。这使得 FPU 对浮点数据使用另外的（可能更高精度的）内部表示来实现传送指令而不用负担来回转换的开销。

即使在 MIPS I FPU 浮点处理器上，这些指令也可以指定奇数号的浮点寄存器：

在整数和浮点寄存器之间：

```
mtc1 s, fd        fd = s; /* 32b uninterpreted */
mfc1 d, fs        d = fs;
dmtc1 s, fd       fd = (long long) fs; /* 64 bits */
dmfc1 d, fs       d = (long long) fs;
```

在浮点寄存器之间：

```
mov.d fd, fs       fd = fs;
                   /* move 64b between register pairs */
mov.s fd, fs       fd = fs; /* 32b between registers */
```

条件传送（MIPS III 及以前都没有）——省略了 .s 版本以节省空间：

```
movt.d fd,fs,cc       if (fpcondition(cc)) fd = fs;
movf.d fd,fs,cc       if (!fpcondition(cc)) fd = fs;
movz.d fd,fs,t        if (t == 0) fd = fs;
                      /* t is an integer register */
movn.d fd,fs,t        if (t != 0) fd = fs;
```

叫做 `fpcondition(cc)` 的浮点条件码难免要向前引用；更多内容参见第 7.9.7 节。如果你想了解条件传送指令有什么用，参见第 8.5.3 节。

7.9.3 三操作数算术运算

要注意以下几点：

- 所有的算术运算都可以引发任一种 IEEE 异常类型，如果硬件对操作数不满意可以导致一个未实现的自陷。
- 所有指令都有单精度（32 位，C `float`）和双精度（64 位，C `double`）两个版本；不同指令通过在操作码上加上“.s”或“.d”来区分。我们只给出双精度的版本。注意不可以混合两种格式：源和目标必须都为单精度或者都为双精度。要混合单精度或者双精度，你需要进行明确的转换操作。

  在所有的 ISA 版本中：

```
add.d   fd, fs1, fs2     fd = fs1 + fs2;
div.d   fd, fs1, fs2     fd = fs1 / fs2;
mul.d   fd, fs1, fs2     fd = fs1 x fs2;
sub.d   fd, fs1, fs2     fd = fs1 - fs2;
```

  下面的指令在 MIPS III 及以前的 CPU 中没有，运算速度快但不完全符合 IEEE 精确度要求——误差最多可达尾数的最低两位所代表的大小（2 ULP）：

```
recip.d fd, fs        fd = 1 / fs;
rsqrt.d fd, fs        fd = 1 / (squarerootof(fs));
```

7.9.4 乘加操作

这是为了响应 SGI 在很高端的图形系统中达到类似超级计算机的性能的号召（与 1995 年 SGI 收购 Cray Research Inc 有关）而在 MIPS III 之后加上的。IBM 的 POWERPC 似乎也从其乘加操作获得了高速优良的浮点性能。尽管由单个指令完成两个工作有违 RISC 的原则，合成的乘加操作在常见的重复的浮点运算中得到了广泛使用（典型的用于矩阵或者向量运算）。此外，通过避免对中间结果的舍入和重新规格化步骤（IEEE 规定将结果写回寄存器时必须如此）而节省了显著的时间。

乘加操作有多种形式，全都需要三个寄存器操作数和一个单独的结果寄存器：

```
madd.d  fd, fs1, fs2, fs3        fd = fs2 x fs3 + fs1;
msub.d  fd, fs1, fs2, fs3        fd = fs2 x fs3 - fs1;
nmadd.d  fd, fs1, fs2, fs3       fd = - (fs2 x fs3 + fs1);
nmsub.d  fd, fs1, fs2, fs3       fd = - (fs2 x fs3 - fs1);
```

IEE 754 并没有具体规定乘加运算，所以为了符合标准产生的结果应当和相乘然后相加两条指令的计算结果完全相同——这里要特别小心! 因为每个浮点操作都会有一些舍入误差，这意味着 IEEE 754 规定的精度对于乘加运算比实际可以达到的要差。

7.9.5 单目（变号）操作

尽管名义算作算术运算，这些操作其实只是改变符号位，所以不会产生大多数的 IEEE 异常。如果给的操作数是个明令 NaN 值，会产生无效自陷。如下所列：

```
abs.d   fd, fs        fs = abs(fs)
neg.d   fd, fs        fs = - fs
```

7.9.6 数据转换操作

注意“从单精度转换到双精度”写作“cvt.d.s” ——与通常一样，先写目标寄存器。转换在浮点寄存器中的数据之间进行：当转换来自 CPU 整数寄存器的数据时，从浮点到 CPU 寄存器之间的传送必须和数据转换分别编码。转换操作可能导致在相应上下文中有意义的任何 IEEE 异常。

一开始，所有这些都有同一族指令完成：

```
cvt.x.y     fd, fs
```

其中 x 和 y 分别指定目标和来源寄存器的格式，为以下之一：

- s C `float`，MIPS/IEEE 单精度, 32 位浮点数
- d C `double`，MIPS/IEEE 双精度, 64 位浮点数
- w C `int`，MIPS/IEEE 字, 32 位整数
- l C `long long`，MIPS/IEEE 双字, 64 位整数

指令如下：

```
cvt.s.d fd, fs          /* double fs -> float, leave in fd */
cvt.w.s fd, fs          /* float fs -> int, leave in fd */
cvt.d.l fd, fs          /* long long fs -> double, leave in fd */
```

从浮点转换到整数格式有不止一种合理的方法，具体结果取决于当前的舍入模式（这由在第 7.7 节讲的 **FSCR** 寄存器设置）。但是浮点计算经常想要明确以某种方式舍入到整数（比如上界运算符向上舍入），生成修改和恢复 **FCSR** 的代码太过麻烦。所以除了最早的 MIPS I CPU 之外，都有以明确指定的方式进行舍入的转换指令，而且用得更多。

明确指定舍入方式的到整数的转换：

```
round.x.y   fd, fs      /* round to the nearest */
trunc.x.y   fd, fs      /* round toward zero       */
ceil.x.y    fd, fs      /* round up */
floor.x.y   fd, fs      /* round down */
```

这些指令只有当 x 代表一种整数格式的时候才能有效。

7.9.7 条件分支和测试指令

浮点分支和测试指令是分开的。我们在下面讨论测试指令——-名字是 c.le.s 这种形式，比较两个浮点值并设置相应的 FPU 条件位。

分支指令因此只需测试条件位为真（置位）或者为假（零）。可以选择指定用哪个条件位：

```
bc1t label        if (fpcondition(0)) branch-to-label;
bc1t cc, label    if (fpcondition(cc)) branch-to-label;
bc1f label        if (!fpcondition(0)) branch-to-label;
bc1f cc, label    if (!fpcondition(cc)) branch-to-label;
```

和其它的 MIPS 分支指令一样，每一个都有一个“可能分支”的变体。[9]

```
bc1tl   label    /* branch-likely forms of bc1t ... */
bc1fl   label
```

和其它称为分支的 CPU 指令一样，分支目标 **label** 编码为一个 16 位的有符号整数，代表以字单位从下一条指令加一（流水线的工作方式很奇特）开始的偏移量。如果 **label** 位于 128KB 以外，你就会有麻烦，不得不求助于 **jr** 指令。

MIPS III 之前的包括其在内的 MIPS CPU 在浮点控制/状态寄存器 **FCSR** 中只有一个浮点状态位，叫做“C”。主流的 CPU 还有 7 个位，叫做 **FCC7–1**。如果你在分支或者比较指令中不指定 cc，就隐含地选择了拥有 FCC0 的荣誉称号的老 C 位。这与过去老的指令集版本一致。（如果你对为什么引入这个扩展感兴趣的话，参见第 8.5.7 节）在所有指令集中，cc 都是可选的。

但是在分支之前，你必须适当设置条件位。比较运算如下：

```
c.cond.d fs1,fs2      /* compare fs1 and fs2 and set FCC(0)  */
c.cond.d cc,fs1,fs2   /* compare fs1 and fs2 and set FCC(0)  */
```

在这些指令中，cond 可以是任意 16 个条件助记符之一。助记符有时候是有明显意义的（eq）有时候显得很神秘（ult）。为什么有这么多呢？实际上比较浮点值得时候有四种互不重合的结果：

[9]参见第 8.5.4 节以了解更多的“可能”分支。

```
fs1 < fs2
fs1 == fs2
fs1 > fs2
unordered (fs1, fs2) ll
```

当其中一个操作数为 IEEE NaN 值时，IEEE 标准定义“无序(unordered)”为真。

通过调换操作数的次序或进行反向测试，我们总可以从“小于或等于”合成“大于。” 这样我们可以有三个结果。MIPS 提供了指令来测试这三个条件的任意“或”组合。在此基础之上，每一个测试有两种形式，一种在操作数无序时发生“无效(invalid)” 自陷，另一种从来不发生这种自陷。

我们不需要为象“不等于”这样的条件提供测试；我们测试等于，然后用 **bc1f** 而不是 **bc1t** 分支条件。表 7.4 列出了所有可用的指令名称。

在许多 CPU 实现中，比较指令的结果产生得太晚，分支指令无法在不引起后继指令延迟的情况下执行。在 MIPS III 及以前的 CPU 中，分支指令如果直接在测试指令后运行可能导致误动作。支持老式 CPU 的编译器和汇编器应当能够产生适当的延迟，必要时插入一条 **nop**。

表 7.4: FP 测试指令

“C”置位条件			助记符 无自陷	助记符 自陷
	always false		**f**	**sf**
		unordered(fs1, fs2)	**un**	**ngle**
	fs1 == fs2		**eq**	**seq**
	fs1 == fs2	|| unordered(fs1, fs2)	**ueq**	**ngl**
fs1 < fs2			**olt**	**lt**
fs1 < fs2	||	unordered(fs1, fs2)	**ult**	**nge**
fs1 < fs2	|| fs1 == fs2		**ole**	**le**
fs1 < fs2	|| fs1 == fs2	|| unordered(fs1,fs2)	**ule**	**ngt**

7.10 单精度对浮点指令以及 MIPS-3D ASE

符合 MIPS32/64 规范的浮点单元可以选择实现单精度对浮点指令。这个扩展的基础是一组对打包进一个 64 位寄存器的一对 IEEE 单精度的值的每一半执行两个操作的算术运算。

听起来好像要造一个全新的 FPU。但是其实调整一下双精度浮点 FPU 来提供单精度对运算成本并不高——至少对于象加法、乘法、乘加、测试、传送等等“一次性” 操作是如此。

我们将使用 fs 和 ft 作为源寄存器，fd 作为目的寄存器。方便地说，如果 fs 是一个包含单精度对值的浮点寄存器，那么 fs.upper 和 fs.lower 分别代表高位和低位的单精度值。

要找出你的 CPU 是否实现了单精度对指令，看一下 **FIR(PS)** 是否置位就知道了。

7.10.1 异常和单精度对浮点指令

通常在你需要用单精度对浮点指令的地方，你可能需要不辞辛苦地防止异常。只要任何一条浮点指令导致异常，那么整个指令都遭受异常：如果其中一条指令需要 MIPS 自陷，那么你得处理一个 MIPS "异常"，指令的两半都要回滚，就象从未开始执行一样。异常交付时没有表明到底是高半部分还是低半部分计算的问题，还是两部分都有问题。

7.10.2 单精度对三操作数算术运算、乘加、变号和无条件数据传送操作

这些操作完成的工作和相应的单精度（.s）版本完全相同，不同之处只是执行两次。就是说，一条 **add.ps fd,fs,ft** 指令等价于：

```
fd.upper = fs.upper + ft.upper;
fl.lower = fs.lower + ft.lower;
```

这些都属于 SIMD（单指令、多数据）指令，对于向量类型操作极为有用：在一个时间里完成两个操作。

但是，并非所有指令都有 **.ps** 形式：

- 相应操作实现了，但是操作助记符和单精度运算相同：**add**、**sub**、**mul**、**abs**、**mov**、**neg**。
- 没有单精度对型操作：按照具体要求进行舍入的所有操作 **round**、**trunc**、**ceil**、**floor**。这些操作看上去在向量化的应用软件中并不那么重要。

  除法、倒数和平方根——对应于 **div.s**、**recip.s**、**sqrt.s** 和 **rsqrt** ——没有单精度对的形式（这些函数典型的实现要用到迭代算法，无法做到一次计算两个值）。

  要是有 MIPS-3D 扩展的话，则增加了可以在两步之内计算倒数、或平方根的倒数的指令。这些指令也可用于单精度浮点对 PS 值。参见第 7.10.5 节。

  没有为整数转换指令 **round**、**trunc**、**ceil**、**floor** 提供单精度对形式。这种操作的结果会是双整数对的格式，为此发明一个全新的数据类型好像不值。

7.10.3 单精度对的类型转换操作

要从两个单精度的值构成一个单精度对值，用 **cvt.ps.s fd, fs, ft** 指令：

```
fd.upper = fs;
fd.lower = ft;
```

要从一个单精度对的高半部或低半部提取一个单精度的值采用 **cvt.s.pl fd,fs** (`fd = fs.lower;`) 或者 **cvt.s.pu fd,fs** (`fd = fs.upper;`)。

要把一个单精度对的值重新调整组织成一个新的单精度对，有四个指令可以选择。每条指令从两个单精度对的寄存器中各取一个值，凑成新的一对：

```
pll.ps fd, fs   fd.upper = fs.lower;
                fd.lower = ft.lower;
plu.ps fd, fs   fd.upper = fs.lower;
                fd.lower = ft.upper;
pul.ps fd, fs   fd.upper = fs.upper;
                fd.lower = ft.lower;
puu.ps fd, fs   fd.upper = fs.upper;
                fd.lower = ft.upper;
```

alnv.ps fd,fs,ft,s 把两个单精度值打包成一个单精度对。其主要作用——与非对齐的加载指令结合使用——是辅助软件从单精度数组中一次加载和打包两个单精度值，即使该数组在存储器中的对齐方式使得每一对的次序错开了。

为了便于理解该指令的目的，把单精度对的值想象成把“a”和“b”的单精度值打包到了一起，这里“a”在存储器中的占据最低地址（寄存器中与 a 和 b 关联的位号随着尾端变化）。

通用寄存器 **s** 的低三位的值只可以为 0 或者 4（s 通常是用来从存储器加载一对值的指针）——4 以外的任何值都会导致异常。所以就象下面这样：

```
if ((s & 7) == 0) {
    fd.a = fs.a; fd.b = fs.b;
} else {
    /* s & 7 == 4 */
    fd.a = fs.b; fd.b = ft.a;
}
```

7.10.4 单精度对的测试和条件传送指令

因为每个单精度对寄存器中有两个值，所以比较指令设置两个条件位。你可以指定偶数号的条件位，指令将会用该位和下一个奇数条件位。所以如果你要是写成 **c.eq.ps 2 fs ft**，就等于说：

```
fcc2 = (fs.upper == ft.upper) ? 1 : 0;
fcc3 = (fs.lower == ft.lower) ? 1 : 0;
```

如果你不记得这些神秘的测试条件指什么，请参阅表 7.4。

MIPS-3D（见下文）包括一些可能会用到的一次测试多个条件码的分支指令。

条件传送指令 **movf.ps** 和 **movt.ps**，一次测试两个浮点条件码并在目标地址的上半部分和下半部分上同时执行两个独立的条件为真即传送的操作，即：

```
movt.ps fd,fs,2     if (fcc2) fd.upper = fs.upper;
                    if (fcc3) fd.lower = fs.lower;
```

与此相对照的是，**movn.ps** 和 **movz.ps** 根据整数寄存器的值进行条件传送，对两半进行同样处理。

7.10.5 MIPS-3D 指令

MIPS-3D ASE（指令集扩展）增加了一些被认为在高质量 3-D 图形坐标计算中能够提高单精度对扩展效率的指令。

- 浮点“归并加/乘”（加/乘对）：
  把一个寄存器内的一对浮点数相加或者相乘。真正的 SIMD 风格，应当是把两个不同寄存器内的单精度对同时相加或相乘：

  addr.ps fd,fs,ft 指令执行的操作为：

  ```
  fd.upper = fs.upper + fs. lower;
  fd.lower = ft.upper + ft. lower;
  ```

  mulr.ps fd,fs,ft 执行：

  ```
  fd.upper = fs.upper * fs. lower;
  fd.lower = ft.upper * ft. lower;
  ```

- 对绝对值（即忽略符号位）的比较操作：
  有一组完整的 **cabs.xx.x** 操作：

cabs.eq.d	**cabs.nge.d**	**cabs.ole.s**	**cabs.ueq.s**
cabs.eq.ps	**cabs.nge.ps**	**cabs.olt.d**	**cabs.ule.d**
cabs.eq.s	**cabs.nge.s**	**cabs.olt.ps**	**cabs.ule.ps**
cabs.f.d	**cabs.ngl.d**	**cabs.olt.s**	**cabs.ule.s**
cabs.f.ps	**cabs.ngl.ps**	**cabs.seq.d**	**cabs.ult.d**
cabs.f.s	**cabs.ngl.s**	**cabs.seq.ps**	**cabs.ult.ps**
cabs.le.d	**cabs.ngle.ps**	**cabs.seq.s**	**cabs.ult.s**

cabs.le.ps	**cabs.ngt.d**	**cabs.sf.d**	**cabs.un.d**
cabs.le.s	**cabs.ngt.ps**	**cabs.sf.ps**	**cabs.un.ps**
cabs.lt.d	**cabs.ngt.s**	**cabs.sf.s**	**cabs.un.s**
cabs.lt.ps	**cabs.ole.d**	**cabs.ueq.d**	
cabs.lt.s	**cabs.ole.ps**	**cabs.ueq.ps**	

- 测试多个条件码的分支指令：

  bc1any2f、**bc1any2t**、**bc1any4f**、**bc1any4t** 在分支之前测试几个条件码的“或 (OR)。”你可以写一条指令 **bc1any2f N,offset**，其中 N 为 0 到 6 之间的条件码。执行的测试包括：

指令	分支时测试的条件
bc1any2f 2, target	`if (!fcc2 || !fcc3) goto target`
bc1any2t 2, target	`if (fcc2 || fcc3) goto target`
bc1any4f 4, target	`if (!fcc4 || !fcc5 || !fcc6 || !fcc7) goto target`
bc1any4t 4, target	`if (fcc4 || fcc5 || fcc6 || fcc7) goto target`

- 倒数和平方根的计算：这主要是因为缺乏单精度对的除法、倒数、平方根运算（当然这些指令的单精度和双精度的版本总是有的）。

  recip1 fd,fs 是对倒数的一种快速粗略的近似计算。这是在不用“地下”的迭代过程能以合理代价计算得到的最好值——迭代过程对单精度浮点对 SIMD 的收敛性不好。比起用标准的 **recip.s** 和 **recip.d** 指令得到的结果，这个近似要差得多，虽然大家都没有达到 IEEE 的精度。**recip1** 的操作不受当前舍入模式的影响（但是要遵守关于微小的“非规格化”值的产生和使用的控制标志）。

  recip2 fd,fs,ft 根本就不象倒数——其实是一个定制的乘加运算，不用向寄存器加载常数 1 而计算 `1-(fs*ft)`。选中这个函数是因为以下三指令的序列能够进一步提高 recip1 的精度：

```
recip1.s  f1, f0          # f1 =~ 1/f0
recip2.s  f2, f1, f0      # f2 = f0 * (error in f1)
madd.s  f1, f1, f1, f2    # f1 = f1 - f2*f1, a better guess
```

  对单精度数这样的结果已经足够好了（你不可能进一步改进到 IEEE 的精度，除非单独处理某些最低有效位，那样的话计算将很长）。对于双精度数要得到最好的结果，还得要有进一步的 **recip2**/**madd** 序列。

  这点最好的地方在于，相应的单精度对的计算直接能用它，三条指令就能产生一对很好的倒数：

```
recip1.ps f1, f0
recip2.ps f2, f1, f0
madd.ps f1, f1, f1, f2
```

rsqrt1/rsqrt2 指令用类似的方法。在这种情形下，用 **rsqrt2 fd,fs,ft** 计算的误差函数为 `(1-fs*ft)/2`。

这次你需要一条额外的乘法指令来生成更好的单精度浮点对的值（仍达不到 IEEE 精度）：

```
rsqrt1.ps  f1,f0
mul.ps  f2,f1,f0
rsqrt2.ps  f3,f2,f1
madd.ps  f1,f1,f1,f3
```

同样的，要得到高质量的双精度浮点值，还需要再运行一次精化改进步骤。

- 整数对和单精度浮点对之间的数据转换：
  因为整数/浮点之间的转换总是严格在浮点寄存器中进行，既然有单精度浮点对，似乎自然也就应该有 32 位的整数对：但是原始的单精度对指令集中并没有后者。

  要把一个打包的 32 位整数对转换为单精度浮点对（两个域同时转换），用 **cvt.ps.pw fd,fs**；逆操作把一个单精度浮点对转换成一对紧缩的字，就是 **cvt.pw.ps fd,fs** 指令。在两种情况下，都是由当前的舍入模式决定近似结果的舍入法。

7.11 指令时序要求

常规的浮点算术运算指令是互锁的，没有必要为了保证正确性而去插入 **nop** 指令或重新组织代码。为了得到最佳性能，编译器应当适当安排浮点指令以充分利用整数指令和浮点流水线执行在时间上的重叠——但是这样要对具体 CPU 的实现细节非常敏感。

在有些老式（不兼容 MIPS32/64）的 CPU 上某些别的交互不是互锁的，对了保证运算正确，程序必须避免特定的指令序列。在这种情况下，你的编译器、汇编器、或者（最后还有）你的人，即程序员，必须仔细考虑下列情况下的时序问题：

- 对浮点控制和状态寄存器的操作：当修改 **FCSR** 时，要小心流水线。**FSCR** 的域可以影响任意能够并行执行的浮点运算。要保证当你在写 FCSR 的时候，没有活动的（已经开始运算但是还未取出运算结果的）浮点运算。寄存器写回也可能会比较晚，留出一两条指令来分开 **ctc1 rs, FCSR** 和受影响的计算指令是明智的。

- 浮点和通用寄存器之间的数据传送：这些操作完成的时间比较晚，如果后续的指令要用结果，正常情况下要损失一两个周期。在 MIPS II 和更早之

前的 CPU 上你必须避免依赖。

- 浮点寄存器加载：和整数加载一样，生效很晚。后面紧跟的指令不能立即用刚加载的值。
- 测试指令和分支：用 **bc1t**、**bc1f** 指令测试浮点条件位时编码必须很仔细，因为对条件位的测试比你预计的可能要早一个周期。所以条件分支不能紧跟测试指令之后。

7.12 指令时序和速度

所有的 MIPS FPU 对大多数算术指令都要花费多于一个时钟周期，因而流水线变得可见了。流水线的效果可以有三种形式表现出来。

- 遇险：在软件为了能正确工作而必须保证指令分开的地方出现。在 MIPS32/64 CPU 上对用户级浮点代码没有遇险。

  即使在老式的 CPU 上，浮点数据导致的浮点指令间也没有遇险。

- 互锁：当硬件为了保护而等待操作数准备就绪从而延迟使用操作数的情况下发生。适当安排代码可以提高性能。
- 可见的流水线：这个发生在硬件在一条指令完成之前就准备开始另一条指令（前提是之间没有数据依赖）的地方。编译器、坚定的汇编程序员可以写出让流水线保持满负荷运转把硬件的工作能力发挥到极限的代码。

  现代的 MIPS FPU 常常是完全流水线化的，允许在每个时钟起始（对于双精度指令可能是每两个时钟）启动一个新的浮点乘法、加法或者乘加运算。但是如果你要在下一条指令用某条指令的结果的话，就会让程序停下来等待：对 2006 年代的 CPU，可能要等四五个时钟周期。换句话说，浮点乘法有一两个时钟周期的*重复率*但是有四五个时钟的*延迟*。如果能避免在浮点结果准备好之前使用，程序就会跑得快些。

对浮点来说，乘法一般和加减法一样快。除法和开平方指令要慢得多；如果你需要反复除以同一个数，计算出倒数用乘法代替就有意义了。

7.13 即需初始化和使能

复位之后，正常情况下将 CPU 的 SR 寄存器初始化为禁止所有可选的协处理器，包括FPU（即协处理器 1）。**SR(CU1)** 必须置位 FPU 才能工作。现代所有的 CPU 都有 32 个 64 位的浮点寄存器，但有一个 MIPS I 兼容模式，当 **SR(FR)** 清零时只有偶数号的寄存器可以用于数学运算。

你应当读取 FPU 实现寄存器；读出为零则表明没有浮点硬件，那么运行系统时应当关闭 CU1。

一旦使能 CU1，应当用你选用的舍入模式和自陷使能设置控制/状态寄存器 **FCSR**。最近舍入和禁止全部自陷之外的其它选择都很少见。还有一个选项要设置 **SR(FS)** 位让很小的结果返回为零，可以免去一个到仿真程序的自陷。这种做法与 IEEE 并不兼容，但是 MIPS 硬件无法生成规定的非规格化结果。

一旦 FPU 运行起来，你就需要保证在中断和上下文切换时保存和恢复浮点寄存器。因为这样（相对来说）比较花费时间，可以对此进行优化。有些操作系统就是这样做的，做法就是下面的“滞后上下文切换 (lazy context switch)”:

- 运行新任务的时候默认禁止浮点。因为任务现在不能访问 FPU ,当调度和暂停的时候就不需要保存和恢复浮点寄存器。
- 当发生 CU1 不可用的自陷时，标识该任务为浮点用户，返回之前使能浮点。
- 当处于核心态时或被中断例程直接间接调用的软件中时，禁止浮点操作。那么你就可以避免在中断时保存浮点寄存器；只有当你在和使用浮点的任务进行切换时才需要保存或恢复浮点寄存器。

7.14 浮点仿真

有些低成本的 MIPS CPU 及核不带 FPU。这些处理器的浮点功能由软件提供，比硬件大约慢 50–300 倍。软件浮点在浮点极少用到的系统上很有用处。

软件处理浮点有两种方案：

- 软浮点：可以请求某些编译器用软件实现浮点。浮点算术运算可能用一个隐藏的库函数来实现，但是像数据传送、加载、存储等杂务可以用与整数一致的方式处理。
- 自陷和仿真：编译器可以生成常规的浮点指令。CPU 在由浮点仿真程序捕获的每个浮点指令上发生自陷。仿真程序进行指令译码并完成软件请求的操作。仿真程序工作的一部分是用存储器模拟浮点寄存器集。

如这里所述，运行时仿真程序对很小的操作数或生僻的操作也必须能够仿真浮点硬件的行为；因为体系结构有意对浮点硬件的职责含糊不清，仿真程序通常都是完整的。但是，仿真程序写得保证精确兼容 IEEE ，而且考虑偶尔才会调用，所以仿真程序代码更多考虑的是正确性而不是速度。

编译的浮点效率要高得多；仿真程序对每条指令都有一个来自陷处理程序、指令译码、寄存器堆仿真等各方面的不小的开销。但是当然了，你必须重新编译全部程序才能利用其性能。

可能还有一些编译器并不提供软件浮点运算：MIPS 体系结构的历史起源于工作站，而工作站都必须带有浮点硬件。

第 8 章　MIPS 指令集参考大全

第八、九两章是写给想要看懂和生成汇编代码（不管是亲自书写汇编代码还是因为要编写或者修改编译器而间接需要）的程序员的。第 9 章才真正讨论汇编语言程序设计，本章仅仅关心的是汇编语言的指令；粗略的说，如果你只需要能够理解反汇编的输出列表，就可以跳过第 9 章。我们先从一段简单的 MIPS 代码和一段概述开始。

8.1　一个简单的例子

下面是 C 库函数 `strcmp(1)` 的一个实现，该函数比较两个字符串，相等时返回零，如果第一个字符串（按字符串顺序）大于第二个返回正值，否则返回负值。下面是一个直接实现的 C 算法：

```
int strcmp (char *str1, char *str2)
{
    char c1, c2;

    do {
        c1 = *str1++;
        c2 = *str2++;
    } while (c1 != 0 && c2 != 0 && c1 == c2);

    return c1 - c2; /* clever: 0, +ve or -ve as required */
}
```

在汇编代码中，C 函数的两个参数从 a0 a1 寄存器中传送。（要是你忘记了寄存器的命名约定的话，请参考表 2.1；第 11.2.1 节详细讨论 MIPS 的标准调用约定。）像这样简单的例程可以随便使用临时寄存器 **t0** 等等而无需保存和恢复，所以显然选择临时寄存器存放临时变量。该函数有一个返回值，按照约定在返回时要存放进 **v0** 寄存器。那我们现在就看一下：

```
strcmp:
1:
```

```
    lbu     t0, 0(a0)
    addu    a0, a0, 1
    lbu     t1, 0(a1)
    addu    a1, a1, 1

    beq     t0, zero, .t01      # end of first string?
    beq     t1, zero, .t01      # end of second string?
    beq     t0, t1, 1b

.t01:
    subu    v0, t0, t1
    j       ra
```

我们自顶向下逐行分析一下：

- 标号：**strcmp** 是个熟悉的名字标号，在汇编语言中可以定义函数的入口点、中间的分支、甚至数据存储位置。

  .t01 是一个合法的标号；“.”在标号中是合法的，不得与程序别处的 C 命名冲突。

  as1: 是一个数字标号，许多汇编器把它当做局部标号。你在程序中可以有许多标号都叫做 1 ；“1f” 指下一个标号，“1b”指上一个。这点非常有用，因为不用为每个仅在局部使用的标号都发明一个新的唯一名字。

- 寄存器名：这里给出的“没钱的”（即不带美元符号的）名字是通常的用法，但是需要汇编代码在到达真正的汇编器之前先要经过某种宏处理程序；典型的做法是用 C 的预处理器，大多数工具包的预处理方式都是显而易见的。

象这样的函数用汇编语言来写并不值得；编译器做得可能更好。我们后面（第 9.1 节）会看到怎样可以把汇编程序写得比这里的强得多。

8.2 汇编指令及其意义

本节包含一个长长的列表，该表由大多数 MIPS 汇编器支持的合法的汇编指令名称（即助记符）的全体组成，包括了直到 MIPS64（第二版）在内的指令集。经过一些实验和痛苦的思考，我最后决定在这个表中同时包括真正的机器指令和汇编器的合成指令。这样对每条指令，我们将列出以下内容：

- 汇编格式：指令怎样写。
- 生成的机器指令：对于机器指令别名或者展开成一列机器指令的汇编指令，我们用一个“⇒”来表示宏展开并列出展开后的指令序列。

- 功能：描述该指令做什么，为了精炼采用 C 伪代码描述。必要时也用到 C 的类型转换。

我们并不是把指令和操作数的每一种可能的组合都列出来，因为那样太长了。我们并不列出下面的内容：

- 三操作数指令的双操作数形式：例如，MIPS 汇编器允许这样写：

  ```
  addu $1, $2                    # $1 = $1 + $2
  ```

  否则必须写成：

  ```
  addu $1, $1, $2
  ```

  只要有意义，就几乎到处都可以这样写。

- 所有可能的 load/store 地址格式(addr)：MIPS 机器指令总是用一个寄存器的内容加上一个 16 位的有符号的偏移量[1]生成 load/store 操作的地址，比如写成 **lw $1,14($2)**。MIPS 汇编器还支持好几个别的寻址模式的地址格式；最明显的就是 **lw $1, thing** 从汇编代码标号（或者外部的 C 名字）为“thing”处加载数据。细节可参看第 9.4 节。注意并非所有这些模式都对任何需要给定内存地址的指令可用。对于汇编指令我们就只写成 **lw t, addr**，对于机器指令，就写成基址+偏移量的格式。

  汇编器提供的 **la**（加载地址）指令采用同样的寻址方式语法，尽管并不加载或存储任何值——只是在目标寄存器中生成一个地址值。

  当合成某些地址格式时（特别是在存储时），汇编器若需要一个临时演草用的寄存器就直接用 **at**。在那些隐含使用寄存器会有影响的代码（例如，在尚未保存中断前所有寄存器值的中断处理程序中）中工作的程序员需要特别注意：GNU 汇编器有一条 **.set noat** 指示可以防止隐含使用 **at**。

- 指令的立即操作数版本：根据古老的传统，指令中嵌入的常数叫做立即数。MIPS 汇编语言让你可以指定最后一个源操作数为常数。MIPS CPU 提供了一些真正的硬件指令对某些操作支持最多达 16 位的立即数，但是建议你还是书写“根” 助记符。象 **addui** 之类的名字被汇编器识别为合法的助记符；但是可能只有编译器生成的中间代码才需要这样做，当我们讨论机器指令（例如：表 8.6）以及在反汇编列表中你会看到这些“立即数”的形式。

  在汇编语言中，并没有把立即数限制为 16 位。你也不需要记住哪种立即数形式要用有符号或者无符号的值；如果你写一个任意常数，汇编器会帮你自动合成，如第 9.3.2 节所述。

  再次提醒，对于有些复杂的情形，汇编器可能需要使用临时寄存器 **at**。

[1] 总是有人会破例；有一些寄存器加寄存器的地址格式但仅用于对浮点寄存器的 load/store。这样做是为了和浮点代码中的多维数组的组织相一致。

8.2.1 带 U 和不带 U (Non-U) 的助记符

在我们开始之前先提一下，关于指令助记符写法有一个地方特别容易混淆。一个“**u**”后缀在汇编助记符上通常被读作“unsigned（无符号）。”但是并非总是这个意思（至少在没有极大地发挥你的想象力时，不能理解成这个意思）。取决于上下文的不同，一个“**u**”后缀有几个有着微妙差别的意义：

- 溢出时自陷还是不自陷：在大多数算术运算中，U 代表“无溢出检测。” 象不带这个后缀的算术运算 **add** 在结果溢出到位 31（把整数当成有符号数时的符号位）时会引发异常。带后缀的变体 **addu** 对同样的操作数给出同样的结果但是不引发异常。

  C 和 C++ 不支持整数溢出异常，总是使用带“**u**”的形式——与变量是有符号还是无符号没有关系。除非你真的知道有特殊的理由，否则应当总是使用 **addu** 等指令。

- 条件设置：通用测试操作指令 **slt**(set if less than) 和 **sltu** (set if lessthan, unsigned) 当操作数解释为负值时必然产生完全不同的结果。

- 乘除：整数乘法运算产生一个相当于操作数双倍长度的结果，这意味着对于有符号数和无符号数的输入会产生完全不同的结果；因而有两条指令：**mult** 和 **multu**。注意，结果的低位部分，留在 **lo** 寄存器，对于有符号和无符号版本都是相同的；只是处理溢出到 **hi** 部分的方式不同。

  整数除法指令也与符号有关（考虑一下 `0xFFFF FFFE` 除以 2），所以也分 **div** 和 **divu**。对于向右移位指令（右移一位就是除以二）也有同样的变体；但是把这个作为 U 显然太过火了；移位指令叫做 **sra** (shift-right arithmetic，算术右移，适用于有符号数)和 **srl** (shift-right logical 逻辑右移)。这个世界真是个奇妙的地方。

- 寄存器的部分加载：加载比寄存器宽度小的数据必须确定怎么处理寄存器中多余的位。对于 **lbu** 之类的无符号指令，该字节加载进寄存器其余位清零（我们称为*零扩展*）。如果字节值代表一个有符号的数，最高位就会告诉我们是否为负。在这种情况下，我们用 **lb** 指令把符号位复制到寄存器的其余位。这叫做*符号扩展*。

8.2.2 除法助记符

在整数乘除的机器码中，由单独分开的指令分别发起运算和提取结果。汇编器喜欢掩盖这一点，为三操作数格式生成宏扩展指令，同时执行除数为零的检查。这本来没有问题，唯一不幸的是，除法的汇编宏指令的名字 **div**，和基本的机器码指令的名字相同。这意味着在汇编中写一条机器除法指令必须要用一些

窍门；采用 **zero** 做目的寄存器的三操作数的汇编除法指令应当只生成机器除法指令。

有些工具链提供了更好的方法摆脱这个混乱，定义一个新的助记符 **divd** (divide direct) 表示硬件除法指令，和 **divo**(divide with overflow check) 表示复杂的宏指令。这个做法还未流行开来，但有可能在某些代码中碰到。

8.2.3 指令清单列表

在汇编描述中，我们用表 8.1 给出的所用的符号约定。表 8.2 按照助记符字母次序给出了一个完整的指令描述清单。

表 8.1: 指令列表中用到的约定

字	用途
s,t	用于操作数的 CPU 寄存器
d	用于接受操作结果的 CPU 寄存器
j	立即数
label	指令流中入口点的名称。
shf	对于平移、环移、提取、插入指令，这是隐含的移位量。
sz	对于提取/插入指令，这代表操作域的宽度。
offs	相对于 PC 的 16 位有符号偏移量，代表以字为单位（每个指令一个字）到一个标号的距离，
addr	用汇编语言写 load/store 指令时可以用的几种合法的数据地址表达式之一（参见第 9.4 节描述汇编器怎样实现不同的选项。）
at	汇编临时寄存器，其实就是 **$1**。
zero	寄存器 **$0**，永远为零值。
ra	返回地址寄存器 **$31**。
hi,lo	通过把 **hi** 和 **lo** 接起来形成双精度整数乘法结果。**hi** 和 **lo** 每个容纳和整数寄存器同样多的位，所以 **hilo** 在 32 位机器上可以容纳 64 位整数，在 64 位机器上可以容纳 128 位结果。
MAXNEG32BIT	分别为用 2 的补码可以表示最小的 32 位和 64 位负数。用 2 的补码表示的一个特性是正数 -MAXNEG32BIT 无法用 32 位表示。
MAXNEG64BIT	用于接受操作结果的 CPU 寄存器
cd	指令写入的协处理器寄存器
cs	指令读取的协处理器寄存器
exception(CAUSE,code)	用于引发 CPU 自陷；CAUSE 决定寄存器域 **Cause(ExcCode)** 的设置。“code”不是一个由硬件解释的值，而是一个编码在指令内的无所谓域，系统软件可以通过读取指令找到该域。并非每个这样的指令都设置“code”值，所以有时就省略了。

表 8.1 续

字	用途
exception(CAUSE)	用于接受操作结果的 CPU 寄存器
const31..16	代表二进制数“const”的位 31 到 16 所得到的数。MIPS 的书也用类似的约定。

表 8.2: 按字母次序排列的汇编指令

汇编/机器码	功能描述
abs d,s ⇒ **sra $at,s,31** **xor d,s,$at** **subu d,d,$at**	`d = s < 0 ? -s : s`
add d,s,j ⇒ **addi d,s,j**	溢出时自陷，很少用 `d = s + (signed)j`
add d,s,t ⇒	溢出时自陷，很少用 `d = s + t`
addu d,s,j ⇒ **addiu d,s,j**	书写时 j 也可以超出 $-32768 \leq j < 32768$ 的范围，但是生成的代码会更复杂。 `d = s + (signed)j`
addu d,s,t	`d = s + t`
and d,s,j ⇒ **andi d,s,j**	仅用于 $0 \leq j < 65535$ — 对于更大的数，要生成额外的指令 `d = s & (unsigned)j`
and d,s,t	`d = s & t`
b label ⇒ **beq** **$zero,$zero,offs**	`goto label`
bal label ⇒ **bgezal $zero,offs**	函数调用（范围有限的相对于 PC 寻址的调用）。注意 **ra** 的返回地址是下下一条指令的地址：下一条指令位于分支延迟槽，在调用开始之前执行过了。
bc0f label **bc0f1 label** **bc0t label** **bc0t1 label**	根据协处理器 0 条件分支。在老的 CPU 上测试 CPU 输入引脚的信号。已经不再是 MIPS32 或 MIPS64 的一部分了。

表 8.2 续

汇编/机器码	功能描述
bc1f $fccN, label **bc1fl $fccN, label** **bc1t $fccN, label** **bc1tl $fccN, label**	根据浮点条件置位/为真(t)或者清零/为假(f) 进行分支；见第 7.9.7 节。现代的 FPU 拥有多个浮点条件位，通过 N=0..7 来选择。老的代码只用条件位 0，这时“$fccN”可以省略。 **bc1fl** 等后缀中的“l”表示这是一条可能分支指令；详见第 8.5.4 节。注意 MIPS32/64 作废了可能分支指令，建议程序员和编译器不要在可能要在多种 MIPS 体系结构的实现上可移植的代码中生成本指令。
bc1any2f $fccN,label **bc1any2t $fccN,label** **bc1any4f $fccN,label** **bc1any4t $fccN,label**	MIPS-3D 指令，根据两个或四个条件的“OR”分支。见第 7.10.4 节。
bc2f label **bc2fl label** **bc2t label** **bc2tl label**	根据协处理器 2 的条件分支。仅当 CPU 使用协处理器 2 指令集或提供外部引脚时才用到。详见上面的 **bc1f** 等等。
beq s,t,label	`if (s == t) goto lable`
beql s,t,label	上面条件分支指令的可能分支变体，已淘汰。仅当分支发生时才执行延迟槽指令；见 8.5.4 节。
beqz s,label ⇒ **beq s,$zero,offs**	`if (s == 0) goto label`
beqzl	beqz 的可能分支变体；见 8.5.4 节。
bge s,t,label ⇒ **slt at,s,t** **beq at,$zero,offs**	`if ((signed) s ≥ (signed) t) goto label`
bgel s,t,label ⇒ **slt at,s,t** **beql at,$zero,offs**	**bge** 的“可能”形式，虽然该指令本身是一个宏。已经完全淘汰，虽然有的汇编器还支持。见第 8.5.4 节。
bgeu s,t,label ⇒ **sltu at,s,t** **beq at,$zero,offs**	`if ((unsigned) s ≥ (unsigned) t) goto label;`
bgez s,label	`if (s ≥ 0) goto label;`
bgezal s,label	如果(s ≥ 0) 调用“`label()`。” 但是注意“返回地址”无条件地保存在寄存器 **ra ($31)** 中。

表 8.2 续

汇编/机器码	功能描述
bgezall s,label	已经淘汰的可能分支变体；见 8.5.4 节。很难看出这条指令有什么好处。
bgezl s,label	已淘汰的可能分支变体；见 8.5.4 节。
bgt s,t,label ⇒ **slt at,t,s** **bne at,$zero,offs**	`if ((signed) s ≥ (signed) t) goto label;`
bgtu s,t,label ⇒ **slt at,t,s** **bne at,$zero,offs**	`if ((unsigned) s ≥ (unsigned) t) goto label;`
bgt s,t,label ⇒ **slt at,t,s** **bne at,$zero,offs**	`if ((signed) s ≥ (signed) t) goto label;`
bgtz s,label	`if (s > 0) goto label;`
bgtzl s,label	**bgtz** 指令的可能分支版本，已淘汰。见 8.5.4 节。
ble s,t,label ⇒ **sltu at,t,s** **beq at,$zero,offs**	`if ((signed) s ≤ (signed) t) goto label;`
bleu s,t,label ⇒ **sltu at,t,s** **beq at,$zero,offs**	`if ((signed) s ≥ (signed) t) goto label;`
blez s,label	`if (s ≤ 0) goto label;`
blezl s,label	**blez** 指令的可能分支版本，已淘汰。见 8.5.4 节。
blt s,t,label ⇒ **slt at,t,s** **bne at,$zero,offs**	`if ((unsigned) s < (unsigned) t) goto label;`
bltl s,label	已淘汰的可能分支版本。见 8.5.4 节。
bltu s,t,label ⇒ **sltu at,s,t** **bne at,$zero,offs**	`if ((unsigned) s < (unsigned) t) goto label;`
bltz s,label	`if (s < 0) goto label;`
bltzal s,label	如果(s <0) 调用 label()。但是不论调用是否发生，“返回地址”一律无条件保存在寄存器 **ra ($31)** 中。 `if (s < 0) label();`

表 8.2 续

汇编/机器码	功能描述
bltzall s,label	生僻的可能分支变体；见 8.5.4 节。
bltzl s,label	已淘汰的可能分支变体；见 8.5.4 节。
bne s,t,label **bnel s,t,label**	`if (s != t) goto label;` 已淘汰的可能分支变体；见 8.5.4 节。
bnez s,label **bnezl s,label**	`if (s != 0) goto label;` 已淘汰的可能分支变体；见 8.5.4 节。
break code	调试器用的断点指令。**code** 值对硬件没有效果，但是断点异常例程可以通过读取异常原因指令检索到它的值。
cache k,addr	对高速缓存行进行操作，如 4.9 节所述。很老的 MIPS CPU 没有实现该指令，管理高速缓存依赖于针对具体 CPU 的技巧。
cfc1 t,cs **cfc2 t,cs**	将数据从协处理器控制寄存器 **cs** 传送到通用寄存器 **t**。只用于采用辅助控制寄存器集的协处理器，例如浮点单元（协处理器 1）。**cfc0** 指令不属于 MIPS32 规范。
ctc1 t,cs **ctc2 t,cs**	将数据从通用寄存器 **t** 传送到协处理器控制寄存器 **cs**。
clo d,s	对 **s** 中的数当作 32 位计算前导（高位）的为一的位的个数。
clz d,s	对 **s** 中的数当作 32 位计算前导（高位）的为零的位的个数。
dabs d,s ⇒ **dsra $at,s,31** **xor d,s,$at** **dsubu d,d,$at**	64 位版本 `d = s < 0 ? -s, s`
dadd d,s,t	64 位版本；溢出时自陷，很少用 `d = s + (signed)j`
addi d,s,j	64 位版本，溢出时自陷，很少用 `d = s + j`
daddiu d,s,j	64 位立即数加法，更多时候写成 **daddu**。 `d = s + j`
daddu d,s,t	64 位版本 `d = s + t`

表 8.2 续

汇编/机器码	功能描述
dclo d,s	对 s 从寄存器的第 63 位开始计算前导（高位）的为一的位的个数。
dclz d,s	对 s 从寄存器的第 63 位开始计算前导（高位）的为零的位的个数。

ddiv $zero,s,t ⇒
ddiv s,t
	因为我们指定 **$zero** 作为目的操作数，该指令为 64 位硬件除法指令。
	`lo = (long long) s / (long long)t;`
	`hi = (long long) s % (long long)t;`

ddiv d,s,t ⇒
teq t,$zero,0x7
ddiv $zero,s,t
daddiu $at,$zero,-1
bne t,$at,1f
daddiu $at,$zero,1
dsll32 $at,$at,31
teq $t1,$at,0x6
1:
mflo d
	执行被零除和溢出检测的 64 位有符号数除法指令。
	`lo = (long long) s / (long long)t;`
	`hi = (long long) s % (long long)t;`
	`if (t == 0) exception (BREAK, 7);`
	`if (t == -1 && s == MAXNEG64BIT)`
	`/* result overflows */`
	`exception (BREAK, 6);`
	`d = lo;`

ddivu $zero,s,t ⇒
ddivu s,t
	无符号 64 位数硬件除法指令。
	`lo = (long long) s / (long long)t;`
	`hi = (long long) s % (long long)t;`

ddivu d,s,t ⇒
teq t,$zero, 0x7
ddivu s,t
mflo d
	带被零除检验的无符号 64 位数除法。
	`lo = (long long) s / (long long)t;`
	`hi = (long long) s % (long long)t;`
	`if (t == 0) exception(BREAK, 7);`
	`d = lo;`

deret	从 EJTAG 调试异常返回。控制传递到 CP0 寄存器 **DEPC** 所包含的地址处的指令，调试模式位清零。挨着 **deret** 之后的指令并不运行（没有延迟槽指令）。有关 EJTAG 的更多内容见第 12.1 节。
dext d,s,shf,sz	从 64 位寄存器提取位域。**shf** 是位域在 **s** 中的移位到第 0 位所需要的位移量，**sz** 是位域的长度。 `s = d(sz+shlf)..shf;`

表 8.2 续

汇编/机器码	功能描述
dext d,s,shf,sz ⇒ **dextm d,s,shf,sz** **dext d,s,shf,sz** ⇒ **dextu d,s,shf,sz**	当 **shf** 或 s 超过 32 位时汇编器根据需要给出 **dextm** 或 **dextu** 机器码。在正常情况下，只要写 **dext** 然后让汇编器处理。
dextm d,s,shf,sz	当 **sz** 为 32 位或更多时的 **dext** 的机器码。
dextu d,s,shf,sz	当 **shf** 为 32 位或更多时的 **dext** 的机器码。
di d	禁止中断。对状态寄存器的全局中断使能位（**AE(IE)**，见第 3.3.1 节）清零，把原来的 **SR** 值保存到 **d** 当中。该操作是原子性的；采用读/改/写方式的替代方案有可能被半路中断导致混乱。

dins d,s,shf,sz	向 64 位寄存器插入位域。待插入的数据构成 **s** 的低位。**shf** 是数据需要向左移位的位移量，**sz** 是位域的长度。
	`d = d63..(shf+sz) | s(sz)..0 |`
	`(shf > 0 ?  d(shf-1)..0 :  0);`

dins d,s,shf,sz ⇒ **dinsm d,s,shf,sz** **dins d,s,shf,sz** ⇒ **dinsu d,s,shf,sz**	当 **shf** 或 s 超过 32 位时汇编器根据需要给出 **dinsm** 或 **dinsu** 的机器码。在正常情况下，只要写 **dins** 然后让汇编器处理。
dinsm d,s,shf,sz	当 **sz** 为 32 位或更多时的 **dins** 的机器码。
dinsu d,s,shf,sz	当 **shf** 为 32 位或更多时的 **dins** 的机器码。

div $zero,s,t ⇒
div s,t
	32 位有符号数的硬件除法指令；当目的寄存器为 **$zero** 时汇编器并不插入被零除或溢出检测代码。
	`lo = s / t;`
	`hi = s % t;`

div d,s,t ⇒
teq t,$zero,0x7
div $zero,s,t
li $at,-1
bne t,$at,1f
lui $at,0x8000
teq s,$at,0x6
1:
mflo d
	有符号 32 位数除法指令，在被零除和溢出条件下发生异常。
	`if (t == 0)`
	`  exception (BREAK, 7);  /* divide by zero */`
	`lo = s / t; hi = s;`
	`if (t == -1 && s == MAXNEG32BIT)`
	`  exception (BREAK, 6);  /* result overflows */`
	`d = lo;`

表 8.2 续

汇编/机器码	功能描述

divu $zero,s,t ⇒
divu s,t
	/* $zero as destination means no checks */
	lo = s / t;
	hi = s % t;

divu d,s,t ⇒
teq t,$zero, 0x7
divu s,t
mflo d
	无符号除法，但是被零除导致异常：
	if (t == 0) exception(BREAK, 7);
	lo = (unsigned) s / (unsigned)t;
	hi = (unsigned) s % (unsigned)t;
	d = lo;

dla t,addr ⇒ **# various ...**	加载 64 位地址；见第 9.4 节。
dla t,const ⇒ **# biggest case** **lui t,const63..48** **ori t,const47..32** **dsll t,16** **ori t,const31..16** **dsll t,16** **ori t,const15..0**	加载 64 位常数。仅当常数值介于 0x8000 0000 和 0xFFFF FFFF 之间时才要用不同于 li 的助记符，此时 32 ⇒64 位转换规则要求 li 将高 32 位全部填为一。
dmadd16 s,t	仅在 NEC 的 Vr41xx 族 CPU 上有本指令。 (long long) lo = (long long)lo + ((short)s * (short)t);
dmfc1 t,fs **dmfc2 t,fs**	将 64 位数据从协处理器寄存器 **cs** 传送到通用寄存器 **t**。当然仅在带有 64 位寄存器的协处理器上需要和实现。**dmfc1** 用于浮点单元寄存器；**dmfc2** 极为罕见。
dmtc1 t,cs **dmtc2 t,cs**	将 64 位数据从通用寄存器 **t** 传送到协处理器寄存器 **cs**。其它同上面的 **dmfc1**。
dmul d,s,t ⇒ **dmultu s,t** **mflo d**	64 位有符号数乘法指令；**s** 和 **t** 的乘积计算结果为 128 位，不可能溢出。 没有单条机器指令实现三寄存器操作数的 64 位乘法，虽然有对于 32 位操作数有这样的指令（**mul**）。 hilo = s * t; /* with 128-bit precision */ d = lo;

表 8.2 续

汇编/机器码	功能描述

dmulo d,s,t ⇒
dmult s,t
mflo d
dsra d,d,63
mfhi $at
tne d,$at,0x6
mflo d
	有符号数乘法指令，溢出时发生异常。
	`hilo = s * t;  /* with 128-bit precision */`
	`if ((lo >= 0 && hi !=0) || (lo < 0 && hi != -1))`
	`  exception (BREAK, 6);`
	`d = lo;`

dmulou d,s,t ⇒
dmultu s,t
mfhi $at
mflo d
tne $at,$zero,0x6
	无符号数乘法指令，溢出时发生异常。
	`hilo = (long long) s * (long long) t;`
	`if (hi !=0)`
	`  exception (BREAK, 6);`
	`d = lo;`

dmult s,t	64 位有符号数乘法的机器指令，结果存放于 **hilo**。 `hilo = (long long) s * (long long) t`
dmultu s,t	64 位机器乘法指令的无符号数版本。 `hilo = (unsigned long long) s * (unsigned long long) t`
dneg d,s ⇒ **dsub d,$zero,s**	单目运算符取相反数，溢出时自陷—你可能想要下面的 **dnegu**。 `(long long) d = -(long long) s; /* trap on overflow */`
dnegu d,s ⇒ **dsubu d,$zero,s**	`(long long) d = -(long long) s;`

drem d,s,t ⇒
teq t,$zero,0x7
ddiv $zero,s,t
daddiu $at,$zero,-1
bne t,$at,1f
daddiu $at,$zero,1
dsll $at,$at,63
teq s,$at,0x6
1:
mfhi d
	有符号 64 位整数取余指令，执行溢出条件检查。
	`if (t == 0) exception (BREAK, 7);`
	`if (s == MAXNEG32BIT && t == -1)`
	`  exception (BREAK, 6);  /* overflow */`
	`d = (long long) s % (long long) t;`

表 8.2 续

汇编/机器码	功能描述

dremu d,s,t ⇒
teq t,$zero,0x7
ddivu $zero,s,t
mfhi d
	64 位无符号整数取余，执行溢出检查：
	`if (t == 0) exception(BREAK, 7);`
	`d = (unsigned long long) s % (unsigned long long)t;`

dret	过时的异常返回指令，用于现在已过时的 R6000 CPU 和某些“MIPS II”后继产品。
drol d,s,t ⇒ **dnegu $at, t** **drotrv d,s,$at**	64 位循环左移，其中环移量是个变量。 `d = (s << t) | ((unsigned long long)s >> (64 - t));`
drol d,s,j ⇒ **drotr d,s,64-j**	64 位循环左移，其中环移量是个常数。 `d = (s << j) | ((unsigned long long)s >> (64 - j));`
dror d,s,t ⇒ **drotrv d,s,t**	64 位循环右移，其中环移量是个变量。 `d = ((unsigned long long)s >> t) | (s << (64-t));`
dror d,s,j ⇒ **drotrv d,s,j**	64 位循环右移，其中环移量是个小于 32 的常量。 `d = ((unsigned long long)s >> j) | (s << (64-j));`
dror d,s,j ⇒ **drotr32 d,s,j**	64 位循环右移，其中环移量是个超过或等于 32 的常量。 `d = ((unsigned long long)s >> j) | (s << (64-j));`
dsbh d,t	交换寄存器（64 位寄存器有四对）的每对字节。
dshd d,t	交换寄存器（64 位寄存器有两对）的每对半字（16 位）。
dsll d,s,t ⇒ **dsllv d,s,t**	64 位左移，移位量是个变量。 `d = (long long) s << (t % 64);`
dsllv d,s,t	你可以这样来写寄存器左移指令的机器码，但是写成 **dsll** 更好。
dsll d,s,shf	64 位左移，移位量是个小于 32 的常量。 `d = (long long) s << shf;`
dsll d,s,shf ⇒ **dsll32 d,s,shf-32**	64 位左移，移位量是大于等于 32 的常量。 `d = (long long) s << shf; /* 32 <= shf < 63 */`

表 8.2 续

汇编/机器码	功能描述
dsra d,s,t ⇒ **dsrav d,s,t**	64 位右移：按 C 的语义为有符号右移，或称算术移位—在每个位向下移动时，寄存器的高位用第 63 位的值填充，正确实现了以 2 的幂作除数的有符号除法。 `d = (signed long long) s >> (t % 64);`
dsra d,s,shf	64 位算术右移，移位量为一个常数（算术移位的意义见上面的 **dsra d,s,t**）。当常数小于 32 时，就是同名的机器指令。 `d = (signed long long) s >> (t % 64);`
dsra d,s,shf ⇒ **dsra32 d,s,shf-32**	同上，32 ≤ **shf**<63。你可能不会想直接写 **dsra32**。
dsrl d,s,t ⇒ **dsrlv d,s,t**	逻辑右移—在每个位向下移动时，寄存器的高位用第 0 填充，和 C 语言无符号整数语义一致。这是移位量为变量的版本。 `d = (unsigned long long) s >> (t % 64);`
dsrl d,s,shf	逻辑右移，移位量为一个小于 32 的常数。 `d = (unsigned long long) s >> (shf % 32);`
dsrl d,s,shf ⇒ **dsrl32 d,s,shf-32**	同上，**shf** 大于或等于 32。
dsub d,s,t	64 位减法，溢出时发生异常，极少见。 `d = s - t;`
dsubu d,s,t	`d = s - t; /* 64-bit */`
ehb	执行遇险防护—当你需要保证上面指令的任何协处理器 0 的副作用在随后的指令执行之前已经完成的时候所用的指令。见第 8.5.10 节。
ei d	中断使能—至少无条件地设置状态寄存器的中断允许位（**SR(IE)**，见 3.3.1 节）。**SR** 原来的值保存到 **d** 中。该操作是原子操作。与上面的 **di** 指令类似（但更有用）。
eret	从中断返回：属于特权模式指令。清除 **SR(EXL)** 位并且分支到 **EPC** 保存的位置去。见第 5.5 节。

表 8.2 续

汇编/机器码	功能描述

ext d,s,shf,sz	从 32 位寄存器提取位域。**shf** 是位域在 **s** 中的移位到第 0 位所需要的位移量，**sz** 是位域包含的位的个数。
	`mask = (2**sz - 1) << shf;`
	`d = (s & mask) >> shf;`

ins d,s,shf,sz	向 64 位寄存器插入位域。待插入的数据构成 **s** 的低位。**shf** 是数据需要向左移位的位移量，**sz** 是位域的宽度。
	`mask = (2**sz - 1) << shf;`
	`d = (d & mask) | ((s << shf) & mask);`

j label	基本的“go-to”指令。注意被限制只能到达 2^{28} 个字节的“页”内指令。 `goto label;`
j r ⇒ **jr r**	跳转到由寄存器 **r** 指向的指令。这是唯一能够将控制转移到任意地址的方法，因为所有的指令内地址格式跨度都小于 32 位。
jal label	子程序调用，返回地址位于 **$ra($31)**。注意返回地址是下下一条指令的—跟通常的 MIPS 分支一样，紧跟分支后的指令位置为分支延迟槽，那里的指令总是在到达子程序之前执行。
jal d,addr ⇒ **la $at,addr** **jalr d,$at**	跟函数调用一样，但是返回地址置于寄存器 **d** 而不是通常的 **$31**。用 **jalr** 指令合成。在机器码展开式中用 **la** 是骗局，因为 **la** 自身是一个宏指令—但这样做我们可以避免在这里解释寻址方式（见 9.4 节）。
jalr d,s	上面 **jal d,addr** 在地址位于另一个寄存器 **s** 时的变体。你可以写成 **jal** 或 **jalr**。
jal s ⇒ **jalr $ra,s**	如果只指定一个寄存器，就是要调用的地址，返回地址置于通常的 **$ra**。
la d,addr ⇒ **# many options**	加载地址—永远是合成指令，根据 **addr** 写法不同可以生成不同的代码序列。更多的内容见第 9.4 节。
lb d,addr	8 位加载，符号扩展到整个寄存器。 对于本指令和其它的 load/store 指令，可以用多种方式写 **addr** 地址—见 9.4 节—但是 load/store 指令只能计算一个寄存器加上 16 位有符号数的地址。 `d = *((signed char *) addr);`

表 8.2 续

汇编/机器码	功能描述
lbu d,addr	8 位加载，零扩展到整个寄存器。 `d = *((signed char *) addr);`
ld d,addr	64 位加载，若地址未对齐到八字节则发生异常。 `d = *((long long *) addr);`
ldc1 d,addr	64 位加载协处理器 1（浮点）寄存器。常写成 **l.d**，见第 8.3 节。
ldc2 d,addr	64 位加载协处理器 2 寄存器，如果采用了协处理器 2 并且宽度为 64 位的话。
ldl d,addr **ldr d,addr**	双精度“向左/向右”加载—成对使用，这些指令实现一个 64 位未对齐的加载 **uld**；见下文以及第 2.5.2 节。
ldxc1 fd,s(t)	64 位加载协处理器 1（浮点）寄存器，采用双寄存器“索引”寻址。常写为 **l.d** 的形式，见第 8.3 节。 `fd = *((double *)(t+b));`
lh d,addr	16 位加载，符号扩展到整个寄存器。 `d = *((signed short *)addr);`
lhu d,addr	16 位加载，零扩展到整个寄存器。 `d = *((unsigned short *)addr);`
li d,j ⇒ **ori d,$zero,j**	用常数值（“立即数”）加载寄存器。该展开式适用于 $0 \leq j \leq 65535$。
li d,j ⇒ **addiu d,$zero,j**	适用于 $-32768 \leq j < 0$ 的情形。
li d,j ⇒ **lui d,hi16(j)** **ori d,d,lo16(j)**	适用于可以表示为 32 整数的其它 j 值的情形。
ll t,addr **lld t,addr**	连锁加载。分别加载 32 位/64 位，带有连锁副作用；用来和 **sc** 或 **scd** 一起实现无锁的信号量（见第 8.5.2 节）。
lui t,u	上位加载立即数（常数 **u** 符号扩展到 64 位寄存器）。 `t = u << 16;`
lw t,addr	32 位加载，64 位 CPU 上进行符号扩展。 `t = *((int *)addr);`
lwc1 fd,addr	加载单精度浮点数到浮点寄存器—常写作 **l.s**。见第 8.3 节。

表 8.2 续

汇编/机器码	功能描述
lwc2 cd,addr	32 位加载到协处理器 2 寄存器，如果实现了的话。很少见。
lwl t,addr **lwr t,addr**	向左/向右加载一个字。参见下面的 **ulw** 和第 2.5.2 节了解这些指令怎样共同协作完成一个非对齐的 32 位加载操作。
lwu t,addr	32 位零扩展加载，仅见于 64 位 CPU。 `t = (unsigned long long) *((unsigned int *)addr);`
lwxc1 fd,t(b)	采用索引（寄存器 + 寄存器）地址加载 32 位浮点。常写为 **l.s**。见第 8.3 节。 `fd = *((float *)(t+b));`
mad s,t	32 位整数乘法累加，MIPS32 的标准指令。两个寄存器以全精度相乘并累加： `hilo = hilo + ((long long) s * (long long) t);`
madu s,t	同上，但是为无符号运算。 `hilo = hilo + ((unsigned long long) s * (unsigned long long) t);`
madd d,s,t **maddu d,s,t**	32 位整数乘法累加，MIPS32 的标准指令。两个寄存器以全精度相乘并累加： `hilo = hilo + ((long long) s * (long long) t);`
madd16 s,t	32 位整数乘法累加，MIPS32 的标准指令。两个寄存器以全精度相乘并累加： `hilo = hilo + ((long long) s * (long long) t);`
mfc0 t,cs **mfc1 t,fs** **mfc2 t,cs**	把32 位数据从协处理器寄存器 **cs** 传送到通用寄存器 **t**—如果 **cs** 是 64 位宽度，则传送的是低 32 位。访问 CPU 控制寄存器离不开 **mfc0**，把浮点单元数据放回到整数寄存器离不开 **mfc1**。**mfc2** 仅当实现了协处理器 2 的时候才有用，这种情况很少见。
mfhc1 t,cs **mfhc2 t,fs**	将 64 位协处理器寄存器 **cs** 或 **fs** 传送到通用寄存器 **t**。 仅当一个 MIPS32 整数单元配置了个 64 位、其它方面兼容 MIPS64 的协处理器时才提供。例如，MIPS 公司的 24Kf 核拥有一个 64 位浮点单元。

表 8.2 续

汇编/机器码	功能描述
mfhi d **mflo d**	将整数乘法单元的运算结果传送到通用寄存器 **d**。**lo** 包含除法的商，以及乘积的低 32 位，或者 **dmul** 乘积的低 64 位。**hi** 包含除法的余数或者乘积的高位。这些指令永远是互锁的；即使最早期的 MIPS CPU 上，硬件也要等待未完成的乘除指令结束。
move d,s ⇒ **or d,s,$zero**	`d = s;`
movf d,s,$fccN	各种各样的条件传送指令—更多内容参见第 8.5.3 节。 `if (!fcc(N)) d = s;`
movf d,s,t	`if (t) d = s;`
movt d,s,$fccN	`if (fcc(N)) d = s;`
movt.d fd,fs,N **movt.s fd,fs,N**	双精度和单精度版本。 `if ((fcc(N)) fd = fs;`
movz d,s,t	`if (!t) d = s;`
msub s,t **msubu s,t**	整数乘法/累加的负版本，分别为有符号和无符号数形式。 `hilo = hilo - ((long long) s * (long long) t);`
mtc0 t,cd **mtc1 t,fd** **mtc2 t,cd**	把32 位数据从通用寄存器 **t** 传送到协处理器寄存器 **cd**。注意这条指令并不遵循先写目标寄存器的通常习惯。 **mtc0** 用于访问 CPU 控制寄存器，**mfc1** 用于把整数单元数据放到浮点寄存器（更多的时候数据直接从存储器加载）。仅当 CPU 采用协处理器 2 指令的时候（这种情况很少见）才实现 **mtc2**。 如果协处理器寄存器是 64 位宽度，数据加载进低位低位，但是高 32 位的状态没有定义。
mthc1 t,cs **mthc2 t,fs**	将 32 位通用寄存器 **t** 传送到 64 位协处理器寄存器 **cd** 或 **fd** 的高位，保留低位不变。 仅当一个 MIPS32 整数单元配置了个 64 位、其它方面兼容 MIPS64 的协处理器时才提供。例如，MIPS 公司的 24Kf 核拥有一个 64 位浮点单元。

表 8.2 续

汇编/机器码	功能描述
mthi s **mtlo s**	将通用寄存器 **d** 的内容分别传送到整数乘法单元的结果寄存器 **hi** 和 **lo**。这点看上去好像没有什么用，但是当从异常返回时恢复 CPU 的状态必不可少。

mul d,s,t	真正的三寄存器 32 位整数乘法，在 MIPS32 中定义，在某些早期的 CPU 上也有。完整的全精度结果依然交付给内部的 **hilo** 寄存器。没有无符号版本。
	`hilo = (long long) s * (long long) t;`
	`d = lo;`

mul d,s,t ⇒ **mult s,t** **mflo d**	当汇编器生成 MIPS32 之前的指令集时可以合成三寄存器乘法。

mul d,s,t ⇒
mult s,t
mflo d
sra d,d,31
mfhi $at
tne d,$at,0x6
mflo d
	带有溢出检查的 32 位有符号乘法。对溢出的监测是通过察看 **hi** 是否简单的为 **lo** 的符号扩展来做到的。
	`hilo = (signed) s * (signed) t;`
	`if ((lo >= 0 && hi != 0) || (s<0) && hi != 1)`
	`  exception(BREAK, 6);`

mulou d,s,t ⇒
multu s,t
mfhi $at
mflo d
tne $at,$zero,0x6
	带有溢出检查的 32 位有符号乘法。
	`hilo = (signed) s * (signed) t;`
	`if ((lo >= 0 && hi != 0) || (s<0) && hi != 1)`
	`  exception(BREAK, 6);`

mult s,t	`hilo = (signed)s * (signed)t;`
multu s,t	`hilo = (unsigned)s * (unsigned)t;`
neg d,s ⇒ **sub d,$zero,s**	这个版本在溢出时自陷，极为少用。 `d = - s;`
negu d,s ⇒ **subu d,$zero,s**	没有溢出，C 语言总是生成这一种。 `d = - s;`
nop ⇒ **sll $zero,$zero,$zero**	空操作，指令码为全零。
nor d,s,t	与别的逐位操作一样，没必要为 64 位 CPU 来个单独的版本。 `d = ~(s | t);`

表 8.2 续

汇编/机器码	功能描述
not d,s ⇒ **nor d,s,$zero**	d = ~s
nudge addr **nudgex s(t)**	预取指令 **pref nudge** 和 **prefx nudge** 的简写形式。见下文和第 8.5.8 节。
or d,s,t	d = s | t;
ori d,s,j	跟一个常数执行“或”操作 OR。为机器指令，常写成 **or d,s,j**: d = s | (unsigned) j;
pref hint,addr **pref hint,t(b)**	针对访存进行优化的预取指令。事先知道可能需要的数据的程序可以进行安排，让所需数据提前进入高速缓存而不产生副作用。具体实现总是可以把 **pref** 当成空操作。 **hint** 定义这是哪种类型的预取；见第 8.5.8 节。 **prefx** 只用于浮点，匹配浮点加载/存储指令所用的寄存器 + 寄存器寻址模式。
r2u s	LSI ATMizer-II 特有的指令；转换成奇怪的浮点格式。结果存于 **lo**。
radd s,t	LSI ATMizer-II 特有的指令；奇怪的浮点加法。结果存于 **lo**。
rdhwr d,$cs	读取硬件寄存器：允许非特权的用户态软件读取一组 CPU 寄存器之一。见第 8.5.12 节。
rdpgpr d,s	从上一个影子寄存器组读取寄存器的值—见第 5.8.6 节。

rem d,s,t ⇒
teq t,$zero,0x7
div $zero,s,t
li $at,-1
bne t,$at,1f
lui $at,0x8000
teq s,$at,0x6
1:
mfhi d
	有符号整数取余指令，执行被零除和溢出条件检查。
	if (t == 0) exception (BREAK, 7);
	if (s == MAXNEG32BIT && t == -1)
	exception (BREAK, 6); /* overflow */
	d = s % t;

表 8.2 续

汇编/机器码	功能描述

remu d,s,t ⇒
teq t,$zero,0x7
divu $zero,s,t
mfhi d
	无符号整数取余，执行被零除检查：
	`if (t == 0) exception(BREAK, 7);`
	`d = (unsigned) s % (unsigned) t;`

rfe	MIPS32 之前（其实只有 MIPS I）当从异常返回时用来恢复 CPU 状态的指令。现在已经过时，此处不再详细介绍。
rmul s,t	LSI ATMizer-II 特有的指令；奇怪的浮点乘法。结果存于 **lo**。
rol d,s,shf ⇒ **rotr d,s,32-shf**	循环左移，其中环移量是个常数。**rotr** 是 MIPS32R2 新增的；对于老的 ISA，本指令由更长的机器指令序列合成。 `d = s << shf) | ((unsigned)s >> (32 - shf);`
rol d,s,t ⇒ **negu $at,t** **rotrv d,s,$at**	循环左移。在 MIPS32R2 之前的指令集缺乏循环右移的机器指令；对于老的指令集，需要更多的机器指令来合成。 `d = (s << t) | ((unsigned)s >> (32-t));`
ror d,s,shf ⇒ **rotrv d,s,shf**	循环右移常数位。在 MIPS32R2 之前的指令集上为合成指令。 `d = ((unsigned)s >> shf) | (s << (32-shf));`
ror d,s,t ⇒ **rotrv d,s,t**	循环右移，环移量是个变量。在 MIPS32R2 之前的指令集上为合成指令。 `d = ((unsigned)s >> t) | (s << (32-t));`
rotr d,s,shf	循环右移，环移量是个立即量：仅在 MIPS32R2 中才有的机器指令。
rotrv d,s,t	循环右移，环移量是个变量：仅在 MIPS32R2 中才有的机器指令。
rsub s,t	LSI ATMizer-II 特有的指令；奇怪的浮点减法。结果存于 **lo**。
sb t,addr	`*((char *)addr) = t;`
sc t,addr **scd t,addr**	条件存储字/双字；见第 8.5.2 节解释。
sd t,addr	如果 **addr** 没有对齐到八字节就会导致异常。 `*((long long *)addr) = t;`

表 8.2 续

汇编/机器码	功能描述
sdbbp c	调试断点指令—不同于 **break** 指令，因为它直接落入调试模式而非异常模式。可选的 **c** 编码到指令中去，供调试器读取。 注意对这条指令有两种不同的编码—已经过时的那种现在仅有 Toshiba 3900 核及其后继产品使用。 见第 12.1 节有关 EJTAG 调试单元的描述。
sdc1 ft,addr	存储浮点双精度寄存器到存储器。常写成 **s.d**。
sdc2 cs,addr	存储 64 位协处理器 2 寄存器到存储器。
sdl d,addr **sdr d,addr**	双精度“向左/向右”存储；见第 2.5.2 节的解释。
sdxc1 fs,s(t)	存储双精度浮点寄存器，采用双寄存器“索引”寻址。常写为 **s.d** 的形式。 `*((double *)(t+b)) = fs;`
seb d,s	寄存器内把字节符号扩展到整个寄存器。 `d = (long long)(signed char)(s && 0xff);`
seh d,s	寄存器内把半字符号扩展到整个寄存器。 `d = (long long)(signed short)(s && 0xffff);`
seq d,s,t ⇒ **xor d,s,t** **sltiu d,d,1**	“条件设置”汇编助记符组中的第一个，模拟真实机器指令 **slt** 构造。 `d = (s == t) ? 1 : 0;`
sge d,s,t ⇒ **slt d,s,t** **xori d,d,1**	`d = ((signed)s >= (signed)t) ? 1 : 0;`
sgeu d,s,t ⇒ **sltu d,s,t** **xori d,d,1**	`d = ((unsigned)s >= (unsigned)t) ? 1 : 0;`
sgt d,s,t ⇒ **slt d,s,t**	`d = ((signed)s > (signed)t) ? 1 : 0;`
sgtu d,s,t ⇒ **sltu d,s,t**	`d = ((unsigned)s > (unsigned)t) ? 1 : 0;`
sh t,addr	存储半字： `*((short *)addr) = t`
sle d,s,t ⇒ **slt d,s,t** **xori d,d,1**	`d = ((signed)s <= (signed)t) ? 1 : 0;`

表 8.2 续

汇编/机器码	功能描述
sleu d,s,t ⇒ **sltu d,s,t** **ori d,d,1**	d = ((unsigned)s <= (unsigned)t) ? 1 : 0;
sll d,s,shf	d = s << shf; /* 0 <= shf <32 */
sll d,t,s ⇒ **sllv d,t,s** **sllv d,t,s**	d = t << (s % 32);
slt d,s,t	d = ((signed) s < (signed) t) ? 1: 0;
slt d,s,j ⇒ **slti d,s,j** **slti d,s,j**	/* j constant */ d = ((signed)s < (signed)j) ? 1 : 0;
sltiu d,s,j	/* j constant */ d = ((unsigned)s < (unsigned)j) ? 1 : 0;
sltu d,s,t	d = ((unsigned)s < (unsigned)t) ? 1 : 0;
sne d,s,t ⇒ **xor d,s,t** **sltu d,$zero,d**	d = (s != t) ? 1 : 0;
sra d,s,shf	32 位右移，移位量为一个常数。按 C 的语义为有符号右移，或称算术移位—在每个位向下移动时，寄存器的高位用第 63 位的值填充，正确实现了以 2 的幂作除数的有符号除法。 d = (signed) s >> shf;
sra d,s,t ⇒ **srav d,s,t**	32 位算术右移，移位量为一个变量： d = (signed) s >> (t % 32);
srl d,s,shf	32 位逻辑右移：就好象 C 的无符号量的移位，寄存器的高位用第 0 填充。 d = (unsigned) s >> shf;
srl d,s,t ⇒ **srlv d,s,t**	32 位逻辑右移，移位量放在寄存器中。 d = (unsigned) s >> (t % 32);
ssnop ⇒ **sll $zero,$zero,1**	“超标量”空操作；是个空操作，但是在同一时钟周期 CPU 不得发送其它指令。用于神秘的时序目的。
standby	进入断电模式之一，仅适用于 NEC Vr4100 族 CPU；**wait**—见下—指令用得更为广泛。

表 8.2 续

汇编/机器码	功能描述
sub d,s,t	溢出时自陷，很少用到。 `d = s - t;`
subu d,s,j ⇒ **addiu d,s,-j**	`d = s - j;`
subu d,s,t	`d = s - t;`
suspend	进入 Vr4100 CPU 的断电模式。
sw t,addr	存储一个字。 `*((int *)addr) = t;`
swc1 ft,addr	存储单精度浮点数到内存—常写作 **s.s**。
swc2 cd,addr	存储协处理器 2 寄存器的 32 位数据，很少见。
swl t,addr **swr t,addr**	向左/向右存储一个字。参见第 2.5.2 节。
swxc1 fs,t(b)	采用（双寄存器）索引地址存储 32 位单精度浮点；常写为 **s.s**。 `*((float *)(t+b)) = fs;`
sync	数据存取防护指令，主要用于多处理器；见第 8.5.9 节。
synci	让 I-cache 与 D-cache 同步。在写完指令之后，但在执行指令对每个高速缓存行大小的内存块运行指令。
syscall B	产生一个“系统调用”异常。 `exception(SYSCALL, B);`
teq s,t	条件自陷指令：如果相应的条件满足，生成一个 TRAP 异常；本条就是： `if (s == t) exception(TRAP);`
teq s,j ⇒ **teqi s,j**	`if (s == j) exception(TRAP);`
tge s,t	`if ((signed)s >= (signed)t) exception(TRAP);`
tge s,j ⇒ **tgei s,j**	`if ((signed)s >= (signed)j) exception(TRAP);`
tgeu s,t	`if ((unsigned)s >= (unsigned)t) exception(TRAP);`
tgeu s,j ⇒ **tgeiu s,j**	`if ((unsigned) s >= (unsigned) j) exception(TRAP);`

表 8.2 续

汇编/机器码	功能描述
tlbp	TLB 维护指令；见第 6 章。 如果当前 **EntryLo** 中的虚拟页号值与 TLB 一项数据匹配，设置 **Index** 指向该项数据。否则设置 **Index** 为非法值 0x8000 0000（最高位置位）。
tlbr	TLB 维护指令；见第 6 章。 将 **Index** 选中的 TLB 项的信息复制到寄存器 **EntryLo**、**EntryHi1**、**EntryHi0** 和 **PageMask** 中。
tlbwi **tlbwr**	TLB 维护指令；见第 6 章。 用 **EntryLo**、**EntryHi1**、**EntryHi0** 和 **PageMask** 中的数据，分别写入由 **Index**（**tlbwi** 指令）或 **Random**（**tlbwr** 指令）所选中的 TLB 数据项。
tlt s,t	更多的条件自陷。 `if ((signed)s < (signed)t) exception(TRAP);`
tlt s,j ⇒ **tlti s,j**	`if ((signed)s < (signed)j) exception(TRAP);`
tltu s,t	`if ((unsigned)s < (unsigned)t) exception(TRAP);`
tltu s,j ⇒ **tltiu s,j**	`if ((unsigned) s < (unsigned) j) exception(TRAP);`
tne s,t	`if (t != s) exception(TRAP);`
tne s,j ⇒ **tnei s,j**	`if (s != j) exception(TRAP);`
u2r s	LSI ATMizer-II 特有的指令；将无符号数转换成为奇怪的浮点。结果存于 **lo**。
udi0 d,r,s,uc 至 **udi15 d,r,s,uc**	在为用户自定义指令保留的指令编码空间内构建指令。这种指令可以采用三个通用寄存器指令，还可以有用于用户逻辑的 5 位辅助操作码 **uc**。
uld d,addr ⇒ **ldl d,addr** **ldr d,addr+7**	未对齐的双精度加载，由第 2.5.2 节详细介绍的向左加载和向右加载指令合成（仅给出大尾端的例子）。
ulh d,addr ⇒ **lb d,addr** **lbu $at,addr+1** **sll d,d,8** **or d,d,$at**	未对齐的半字加载和符号扩展。这里是按照大尾端展开的（小尾端的情形留给读者作为一个练习）。根据不同的寻址模式，展开后可能比这里的更为复杂。

表 8.2 续

汇编/机器码	功能描述
ulhu d,addr ⇒ **lbu d,addr** **lbu $at,addr+1** **sll d,d,8** **or d,d,$at**	未对齐的半字加载和零扩展。
ulw d,addr ⇒ **ldl d,addr** **ldr d,addr+3**	未对齐的字加载；如果是 64 位则进行符号扩展（只给出大尾端的情形）。见第 2.5.2 节。
usd d,addr ⇒ **sdl d,addr** **sdr d,addr+7**	未对齐的双精度存储。
ush d,addr ⇒ **sb d,addr+1** **srl d,d,8** **sb d,addr**	未对齐的半字存储。
usw d,addr ⇒ **swl d,addr** **swr d,addr+3**	未对齐的字存储；见第 2.5.2 节。
wait	MIPS32 指令，用来进入某种断电状态。通常通过中止执行直到检测到中断实现。软件不应当假定中止总会发生或从 **wait** 唤醒一定表明是非屏蔽的中断—**wait** 指令应当从空转循环中调用。
wrpgpr cd,t	写入前一个影子寄存器组的一个寄存器；详细情况见 5.8.6 节。
wsbh	对 32 位内的两个半字内部进行字节交换。这是一条 32 位指令；在 64 位 CPU 上寄存器的高半部用第 31 位的符号扩展填充。 **wsbh** 与循环移位一起用少数指令实现许多不同形式的字节重组。
xor d,s,t	`d = s ^ t;`
xor d,s,j ⇒ **xori d,s,j**	`d = s ^ j;`

8.3 浮点指令

有一组不多不少的浮点指令（见表 8.3 和 8.4）。但是随着发展很快就呈现出了自己的复杂性。请留意以下几点：

- 每一条浮点指令都有一个单精度版本和一个双精度的版本，在助记符中用 **.s** 和 **.d** 来区分。

  如果你的硬件支持第 7.10 节所述的单精度浮点对扩展，还会有一个 **.ps** 版本的指令，行为和单精度版本完全一样，但是做两次。不属于大表中某个指令的 **.ps** 版本的单精度浮点对指令在表中列了出来，但是我们仅是让你去参考第 7.10 节的说明。

  为了节省空间并且避免让你的眼睛流泪，只要同样的说明适用于两个版本，表 8.4 就只列出单精度版本。

- 基本的浮点编码和指令结果符合 IEEE 754 标准。在硬件不能生成 IEEE 754 标准规定的地方，缺省的行为是引发异常，让软件仿真程序弥补与生成符合标准的结果之间的差距。

- 浮点计算和类型转换指令可以导致异常。这句话在 IEEE 意义下成立，指检测到程序员可能感兴趣的条件；在底层体系结构意义下也成立：MIPS FP 浮点硬件，在面对无法正确处理的操作数和运算的组合时，就会生成一个浮点“未实现” 的异常，此时由让软件仿真程序为它执行浮点运算。

  数据传送指令（存储器、寄存器间传送）从来不会发生异常。**neg.s**、**neg.d**、**abs.s** 或 **abs.d** 指令仅仅改变符号位而不检查内容：唯一会导致自陷的条件是对这些指令给一个“信令 NaN”操作数会产生一个 IEEE “无效 (invalid)”异常。

表 8.4: 按照助记符排序的浮点指令描述

汇编码	功能
abs.s fd,fs	`fd = (fs < 0) ? -fs : fs;`
add.s fd,fs,ft	`fd = fs + ft;`

addr.ps fd,fs,ft	`/* MIPS 3D "reduction add", see Section 7.10 */`
	`fd.upper = fs.upper + fs.lower;`
	`fd.lower = ft.upper + ft.lower;`

alnv.ps fd,fs,ft,rs	将单精度值打包成一个单精度对，根据 **rs** 的值指定的两种可能方式之一进行。

表 8.4 续

汇编码	功能
bc1f $fccN,label **bc1fl $fccN,label** **bc1t $fccN,label** **bc1tl $fccN,label**	几个浮点条件分支指令，表 8.2 都有。
bc1any2f $fccN,label **bc1any2t $fccN,label** **bc1any4f $fccN,label** **bc1any4t $fccN,label**	MIPS-3D 指令，根据两个或四个条件的“或”操作 OR 的结果分支。见 7.10.4 节。
c.eq.s $fccN,fs,ft **c.f.s $fccN,fs,ft** **c.le.s $fccN,fs,ft** **c.lt.s $fccN,fs,ft** **c.nge.s $fccN,fs,ft** **c.ngl.s $fccN,fs,ft** **c.ngt.s $fccN,fs,ft** **c.ole.s $fccN,fs,ft** **c.olt.s $fccN,fs,ft** **c.seq.s $fccN,fs,ft** **c.sf.s $fccN,fs,ft** **c.ueq.s $fccN,fs,ft** **c.ule.s $fccN,fs,ft** **c.ult.s $fccN,fs,ft** **c.un.s $fccN,fs,ft**	浮点比较指令，比较 **fs** 和 **ft** 并把结果存入浮点条件位 **$fccN**。第 7.9.7 节对此有详细介绍。
cabs.xx.s $fccN,fs,ft	MIPS-3D 扩展指令，比较两个浮点值的绝对值并保存比较结果。“**xx**”表达的测试与上面的 **c.xx.s** 指令相同。
ceil.l.d fd,fs **ceil.l.s fd,fs**	将浮点值转换为相等的或下一个有符号的 64 位整数值。
ceil.w.d fd,fs **ceil.w.s fd,fs**	将浮点值转换为相等的或下一个有符号的 32 位整数值。
cfc1 rt,fs **ctc1 rs,fd**	在浮点控制寄存器和通用寄存器之间（“f”表示来自(from) FP，“t”表示送到(to) FP）拷贝数据。用于第 7.7 节介绍的 **FCSR** 寄存器等。
cvt.d.l fd,fs **cvt.d.s fd,fs** **cvt.d.w fd,fs**	浮点类型转换，其中类型 **d,l,s,w**（分别代表 double, long, float, int) 表示目的和来源寄存器的类型。当转换会损失精度时，采用 **FCSR(RM)** 的当前舍入模式来确定执行怎样的近似。对于整数转换，当需要的近似与算法有关时，你最好把指令写成 **floor.w.s** 等形式。

表 8.4 续

汇编码	功能
cvt.l.d fd,fs **cvt.l.w fd,fs** **cvt.s.d fd,fs** **cvt.s.l fd,fs** **cvt.s.w fd,fs** **cvt.w.d fd,fs** **cvt.w.s fd,fs**	
cvt.ps.s fd,fs,ft	将两个单精度值转换成一个单精度对，见第 7.10 节。
cvt.ps.pw fd,fs,ft	MIPS-3D 指令，将两半分别代表一个 32 位整数的值转换为一个单精度对，见第 7.10.4 节。
cvt.pw.ps fd,fs,ft	MIPS-3D 指令，将一个单精度对的两半同时转换为整数值，见第 7.10 节。
cvt.s.pl fd,fs,ft **cvt.s.pu fd,fs,ft**	将一个单精度对的一半转换为通常的单精度值，见第 7.10 节。
div.s fd,fs,ft	`fd = fs / ft;`
dmfc1 rd,fs	将 64 位值从浮点（协处理器 1）不作转换传送到整数寄存器。
dmtc1 rd,fs	将 64 位值不作转换和检查从整数传送到浮点（协处理器 1）寄存器。
floor.l.d fd,fs **floor.l.s fd,fs**	将浮点转换为相等或者下一个较低的 64 位整数。
floor.w.d fd,fs **floor.w.s fd,fs**	将浮点转换为相等或者下一个较低的 32 位整数。
l.d fd,addr ⇒ **ldc1 fd,addr**	加载浮点双精度值，必须对齐到八字节。 `fd = *((double *)(o + b));`
l.s fd,addr ⇒ **lwc1 fd,addr**	加载浮点单精度值，必须对齐到四字节。 `fd = *((double *)(o + b));`
ldc1 fd, disp(b)	已经淘汰的与 **l.d** 等价的指令。
l.d fd,t(b) ⇒ **ldxc1 fd,t(b)**	采用索引寻址加载浮点寄存器。最好写成 asl.d 形式。注意两个寄存器的地位并不完全对称—**b** 用来容纳一个地址而 **t** 为偏移量，如果 **(b+t)** 的值最后落入整个 MIPS 地址映射中不同于 **b** 的区域（由 64 位地址的最高 2 位决定），那么就违反了要求。 `fd = *((double *)(b + t));`

表 8.4 续

汇编码	功能
li.s fd,const **li.d fd,const**	加载浮点常数，通常为合成指令，把常数置于某个存储器位置然后加载。
luxc1 fd,i(b)	双寄存器索引加载双精度浮点数，几乎与 **ldxc1** 一模一样，不同之处在于如果最后地址是非对齐的，不会发生异常，加载将从通过把有效地址的低 3 位清零得到的地址处进行。和 **alnv** 一起用来处理未对齐的单精度浮点对—见第 7.10 节对 **alnv** 的描述。
lwc1 fd, disp(b)	已经淘汰的与 **l.s** 等价的指令。
lwxc1 fd, i(b)	明确的双寄存器索引加载指令；通常比采用相应寻址模式的 **l.s** 要好。见上面关于 **ldxc1** 的注释。
madd.s fd,fr,fs,ft	`fd = fr + fs * ft;`
lwxc1 fd, i(b)	明确的双寄存器索引加载指令；通常比采用相应寻址模式的 **l.s** 要好。见上面关于 **ldxc1** 的注释。
lwxc1 fd, i(b)	明确的双寄存器索引加载指令；通常比采用相应寻址模式的 **l.s** 要好。见上面关于 **ldxc1** 的注释。
madd.s fd,fr,fs,ft	`fd = fr + fs * ft;`
mfc1 rd,fs	将 32 位值从浮点（协处理器 1）不作转换传送到整数寄存器。
mfhc1 rd,fs	将从浮点（协处理器 1）寄存器的高 32 位值传送到整数寄存器。用于带有 64 位 FPU 的 32 位整数 CPU。
mov.s fd,fs	`fd = fs;`
movf.s fd,fs,N	`if (!fcc(N)) fd = fs;`
movn.s fd,fs,N	`if (t != 0) fd = fs;  /* t is a GPR */`
movt.s fd,fs,N	`if (fcc(N)) fd = fs;`
movn.s fd,fs,N	`if (t == 0) fd = fs;  /* t is a GPR */`
msub.s fd,fr,fs,ft	`fd = fs * ft - fr;`
mtc1 rd,fs	将 32 位值不作转换和检查从整数传送到浮点（协处理器 1）寄存器。
mthc1 rd,fs	将 32 位值从整数寄存器传送到浮点（协处理器 1）寄存器的高 32 位。主要用于带有 64 位 FPU 的 32 位整数 CPU（MIPS CPU 的大多数都是 64 位）。

表 8.4 续

汇编码	功能
mul.s fd,fs,ft	fd = fs * ft;

mulr.ps fd,fs	/* MIPS-3D "Reduction Add", see Section 7.10.4 */
	fd.upper = fs.upper * fs.lower;
	fd.lower = ft.upper * ft.lower;

neg.s fd,fs	fd = -fs;
nmadd.s fd,fr,fs,ft	fd = -(fr + fs * ft);
nmsub.s fd,fr,fs,ft	fd = -(fr - fs * ft);
pll.ps fd,fs,ft **plu.ps fd,fs,ft** **pul.ps fd,fs,ft** **puu.ps fd,fs,ft**	重新打包单精度对，见第 7.10 节。
prefx hint, i(b)	寄存器/寄存器寻址模式高速缓存预取指令。仅在浮点上可用（同样的寻址模式也用于 **ldxc1/sdxc1**）。但在整数指令表中也列出了本指令。见第 8.5.8 节介绍的工作机制。
pul.ps fd,fs,ft **puu.ps fd,fs,ft**	按照字母次序的重复—见上面的 **pll**。
recip.s fd,fs	快速求倒数。不满足 IEEE 精度，但是最多只是最低小数位上差一个单位。 fd = 1/fs;
recip1.s fd,fs **recip2.s fd,fs,ft**	MIPS-3D—**recip1** 是个快速粗糙的近似法求倒数，**recip2** 是个进行倒数精化步骤的特殊乘加指令。见第 7.10.4 节。
round.l.d fd,fs **round.l.s fd,fs**	将浮点转换成为相等的或最接近的 64 位整数。
round.w.d fd,fs **round.w.s fd,fs**	将浮点转换成为相等的或最接近的 32 位整数。
rsqrt.s fd,fs	快速还比较精确的（误差不超过最低的两位），但是不满足 IEEE 的精确度。 fd = sqrt(1/fs);
rsqrt1.s fd,fs **rsqrt2.s fd,fs,ft**	MIPS-3D—**rsqrt1** 是个快速粗糙的平方根近似计算指令，**rsqrt2** 是执行平方根精化步骤的一个特殊的乘加运算。见第 7.10.4 节。

表 8.4 续

汇编码	功能
s.d ft,addr ⇒ sdc1 ft,addr	浮点双精度存储，地址必须对齐到八字节。在只带有 32 位 FPU 的 CPU 上需要两条 **swc1** 指令合成。 `*((double *)addr) = ft;`
s.s ft,addr ⇒ swc1 ft,addr	浮点单精度存储，地址必须对齐到四字节。 `*((float *)addr) = ft;`
sdc1 fd, disp(b)	已经淘汰的与 **s.d** 等价的指令。
sdxc1 fd, i(b)	明确的双寄存器索引双精度存储指令—见上面关于 **ldxc1** 的注释，通常比采用相应寻址模式的 **l.s** 要好。
sqrt.s fd,fs	`fd = sqrt(fs);  /* IEEE compliant */`
sub.s fd,fs,ft	`fd = fs - ft;`
suxc1 fd,i(b)	双寄存器索引存储，几乎与 **sdxc1** 一模一样，不同之处在于非对齐的地址不会发生异常，存储照样执行，但是把有效地址的向下舍入到八字节的边界。
swc1 fd, disp(b)	已经淘汰的与 **s.s** 等价的指令。
swxc1 fd, i(b)	明确的双寄存器索引存储 32 位浮点值得指令；通常比采用相应寻址模式的 **s.s** 要好。见上面关于 **ldxc1** 的注释。
trunc.l.d fd,fs **trunc.l.s fd,fs**	将浮点截去小数部分取整转换成为 64 位整数。
trunc.w.d fd,fs **trunc.w.s fd,fs**	将浮点截去小数部分取整转换成为 32 位整数。

8.4 与 MIPS32/64 第一版的差别

MIPS32/64 规范于 2003 年升级到了“第 2 版”，这就是我们这里讨论的。所以本节要做的就是对符合规范第一版的 CPU 中没有的东西做一个概括。

8.4.1 第二版增加的普通指令

这里列出的指令包括几个为 2003 年 MIPS32/64 第二版定义的几条指令。列在这里的目的是为了帮助那些使用符合第一版规范设计的不包含这些指令的 CPU 的人。

表 8.3: 浮点寄存器和标识符约定

用字	用途
fs, ft	浮点寄存器操作数。
fd	接受运算结果的浮点寄存器。
fdhi, fdlo	32 位处理其中一对相邻的浮点寄存器，合在一起存放一个浮点双精度的值。高序号（奇数号）寄存器在正常的算术运算中是隐含的。
$fccN	**FCSR** 寄存器中的浮点条件位之一，**bclt** 之类的指令测试。这些已经有一些发展变化；MIPS III ISA 只有一个条件位，但是现代的 FPU 有 8 个。没有指定用哪个条件位的指令缺省使用原先的“第 0 个”条件位。
`fcc(N)`	同上，用于 C 语言代码。
upper,lower	第 7.10 节讨论的容纳一个单精度浮点对的浮点寄存器的高半部和低半部的值。

提取和插入位域

ext 与 **ins** 指令（以及 64 位变体 **dext**、**dextm**、**dextu**、**dins**、**dinsm** 和 **dinsu**）能够更有效率地访问固定的位域（域的宽度和偏移量内建于这些双寄存器指令里）。

字的重新组织——轮转、字节扩展和交换帮助

为了响应实现更多的寄存器到寄存器间数据重新组织的需求（在网络上应用程序特别关心这一点），加了些新的指令：

- 位轮转操作（即循环移位）：在 MIPS32/64R2 之前，MIPS 只有普通移位操作。现在有了 **rotr**（环移量在指令中编码）和 **rotrv**（环移量由源寄存器指定），还有相应的 64 位版本 **drotr** 和 **drotrv**。
- 字节或者半字的符号扩展：总可以通过向左移位然后向右算术移位实现符号扩展，但指令 **seb**、**seh** 分别提供了符号扩展字节/半字的寄存器到寄存器的操作。
- 半字内的字节交换：**wsbh** 指令看上去有点怪，但和循环移位相结合，可以在一两条指令内实现 32 位字内部有用的大多数字节重组。

在 32 位 CPU 上提供 64 位 FPU（及 CP2）

当 32 位 CPU 配备浮点硬件的时候，配备一个全 64 位的 FPU 常常更有意义，这一点已经变得很明显了。MIPS32/64 第一版允许这样做，但是不经过内存

中转就无法在在通用寄存器和浮点寄存器之间传输全 64 位值。**mfhc1/mthc1** 指令完成这个工作，来回拷贝浮点寄存器的高半部分——已有的 **mfc1/mtc1** 指令已经处理了低半部分。

第二版还定义了 **mfhc2/mthc2** 指令，对 CPU 可能定义的任何 64 位的协处理器 2 完成同样的操作。

读取硬件寄存器

rdhwr 向用户态提供了对某些具体 CPU 信息的只读访问，参见第 8.5.12 节。

让新写入的指令可见

synci 指令完成一切必要的高速缓存操作，使得能够可靠地保证，任何刚刚写入存储器的指令对通过 I-cache 指令高速缓存的取指操作都可见。与老的高速缓存指令不同，这条指令不是特权指令，用户态程序也可以使用。参见第 8.5.1 节。

8.4.2 第二版增加的特权指令

特权（核心专用）指令集也有两处变化。

原子性的中断禁止/允许

在第二版之前，在 MIPS CPU 中没有方法能够原子性地禁止所有中断：在 SR 寄存器上的要求的 RMW 序列本身也可能被中断。你可以通过操作系统的约束（参见第 5.8.3 节）实现安全，但是现在你有了一条真正的原子指令。

di 在单个的原子步骤内禁止中断（清除 **SR(IE)** 位）。把老的 **SR** 值返回到目标寄存器内，所以你一般可以用 **mtc0** 和保存的 **SR** 值重新允许中断。但是为了保持指令集的对称性，第二版还定义了一条 **ei** 指令，原子性地置位 **SR(IE)**。

影子寄存器支持

影子寄存器是通用寄存器集的一组或多组备份，你可以选择在某个异常处理程序中使用，最常见的是在中断处理程序中用。第二版的规范提供了几种不同的使用方法，参见第 5.8.6 节。

涉及到的新指令有 **rdpgpr/wrpgpr**，分别读取和写入当前寄存器集之外某个寄存器组中的一个寄存器。

8.5 特殊指令及其用途

MIPS 从不回避创新，它的指令集里包含一些非常精巧的特性，这些特性因为难以理解或没有解释清楚而不为人所注意（因而就未使用）。本节集中讨论这些特性。

8.5.1 向左加载/向右加载：未对齐的加载和存储

如果常用的数据安排对齐到符合硬件要求的存储器边界效率就会提高。对于具有 32 位总线的的机器来说，就是说将 32 位的数据存放在对齐到 32 位边界的地址有利；类似地，64 位总线对于存放在对齐到 64 位边界的数据有利。

如果 CPU 必须存取穿过存储器宽度边界的非对齐数据，就必须执行两次操作。RISC 流水线的简单性不会允许在一条指令中执行两个操作，所以非对齐的传输至少要花费两个指令。

最极端的 RISC 态度是既然我们有字节大小的操作，想要的非对齐操作可以用字节操作构造出来。如果一块数据（格式为一个四字节或者八字节的整数值）可能是非对齐的，程序员/编译器总能够把它读为一系列字节值的序列，然后用移位、掩码操作在寄存器中重新构建这个数。对字大小的加载序列看上去如下（这里假定是大尾端 CPU，也没有对 CPU 流水线中的加载延迟进行优化）：

```
lbu     rt, o(b)
sll     rt, rt, 24
lbu     rtmp, o+1(b)
sll     rtmp, rtmp, 16
or      rt, rt, rtmp
lbu     rtmp, o+2(b)
sll     rtmp, rtmp, 8
or      rt, rt, rtmp
lbu     rtmp, o+3(b)
or      rt, rt, rtmp
```

这是十条指令，执行四次加载还要用一个临时寄存器。如果这样的操作很多的话，很可能成为性能杀手。

MIPS 对此的解决方案是一对指令，其中每条指令尽可能多地获得非对齐的字位于对齐的内存块中的部分。为 MIPS 指令集发明的这些指令的名字叫做 *向左加载* *向右加载*：一对指令足够进行非对齐的 load/store（字或者双字大小）操作。这在第 2.5.2 节提到过。

访问内存（或者高速缓存）的硬件传输四或八字节的对齐的数据。部分字存储要么通过硬件信号告诉内存控制器保留某些字节不变，要么通过读-改-写（RMW）序列对整个字/双字操作。MIPS CPU 大多数有针对高速缓存的 RMW 硬件，但是没有针对内存的——内存控制器必须自己实现部分写。

我们说要有两条指令，因为有两个总线周期。32 位指令为 **lwl** 和 **lwr**，分别表示“load word left”和“load word right”；64 位指令为 **ldl** 和 **ldr**，分别表示“load double left”和“load double right。” “left” 指令处理未对齐整数的高位，而“right”指令存取低位（“left”的用法和“shift left (向左移位)”中的用法意义相同）。因为指令是用高位和低位定义的，但是必须处理到字节地址寻址的内存，详细的用法取决于 CPU 的尾端（参见第 10.2 节）。一个大尾端的 CPU 先把高位保存在低字节地址，小尾端的 CPU 后把高位保存在高字节地址。

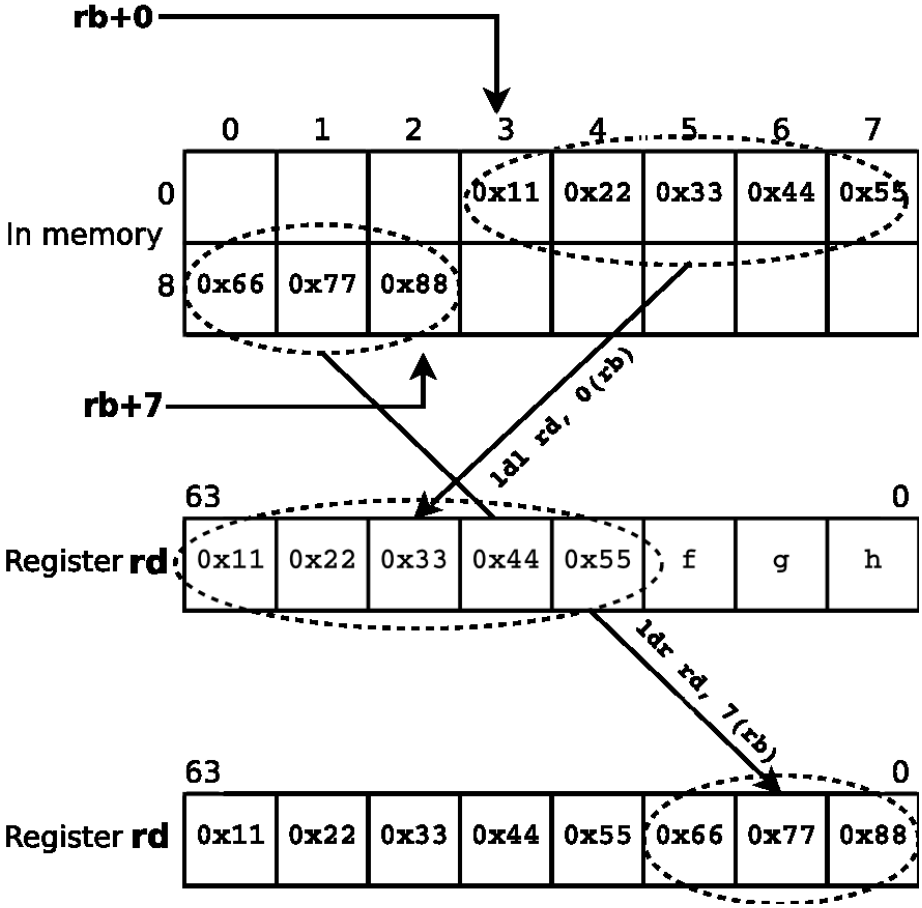

图 8.1: 在大尾端的 CPU 上加载非对齐的双字

图 8.1 试图说明在大尾端 CPU 上执行如下编码的非对齐的宏操作 **uld d, 0(b)** 的整个过程：

```
ldl     d, 0(b)
ldr     d, 7(b)
```

b 中的地址值，当然可以是任意对齐的（非对齐的字用传统的 **ld** 指令会导致异常）。那么图 8.1 究竟在说些什么呢？

- **ldl d, 0(b)**: 0 偏移量标记未对齐双字的最低的字节地址，因为我们是大尾端，这就是最高 8 位。**ldl** 寻找要加载进寄存器左边的位（最高位），所以取得被寻址的字节以及内存中从该字节向后到字的结尾之间的字节。内存地址在增长，所以有效权重向下降；如图所示，这些字节要向上紧接到寄存器的高位编号端。

- **ldl d, 7(b)**: 7 是有点奇怪，但指向双字的最高字节地址——当然 **b+8** 指向下一个双字的第一个字节。**ldr** 关心的是最右边的最低有效位；该指令取出原始数据剩下的字节并把它们紧接寄存器的最低位存放，这样就搞定了。

如果你对这种做法是否对任意对齐的字都有效感到怀疑，那就自己试一试吧。注意在那种所用地址其实刚好已经正确对齐的情形下（这样可以用一条常规指令加载数据），**uld** 对同样数据加载两次；这没有什么特别的意义，但通常也没有什么害处。

这个情况对习惯于小尾端整数序的人显得更为混乱，因为他们书写数据结构的时候经常把最低有效位写在左边。一旦你这样做了，指令名中的“左”就是图中的“右”（当然还是向着高有效位方向移动）。

在小尾端 CPU 上，**ldl/ldr** 的作用正好互换，代码序列如下：

```
ldr     d, 0(b)
ldl     d, 7(b)
```

图 8.2 说明了怎么回事：最高有效位被不情愿的置于左边，所以刚好和我正常画的图构成一对镜像。

面前有了这些图，我们可以对这些指令的行为尽量给出一个精确的叙述：

- Load/store left: 找出被寻址的字节以及包含该字节的字（64 位的话就是双字）。在处于被寻址的字节和该字的最低有效端（大尾端的高字节地址和小尾端的低字节地址）之间的所有字节上操作。

  Load: 取出这些字节移位到寄存器的最高端。保留该寄存器的低位编号的字节不变。

  Store: 只要寄存器中还有空间，用寄存器中从最高有效字节开始的尽可能多的字节取代这些字节。

- Load/store right: 找出被寻址的字节及包含的字/双字。对被寻址的字节以及介于该字节和包含字的最高有效端之间（大尾端的低字节地址，小尾端的高字节地址）的字节进行操作。

  Load: 取得所有这些字节移位到该寄存器的底部。保留寄存器内高字节的内容不变。

  Store: 用从寄存器的最低有效字节开始尽可能多的字节取代这些字节。

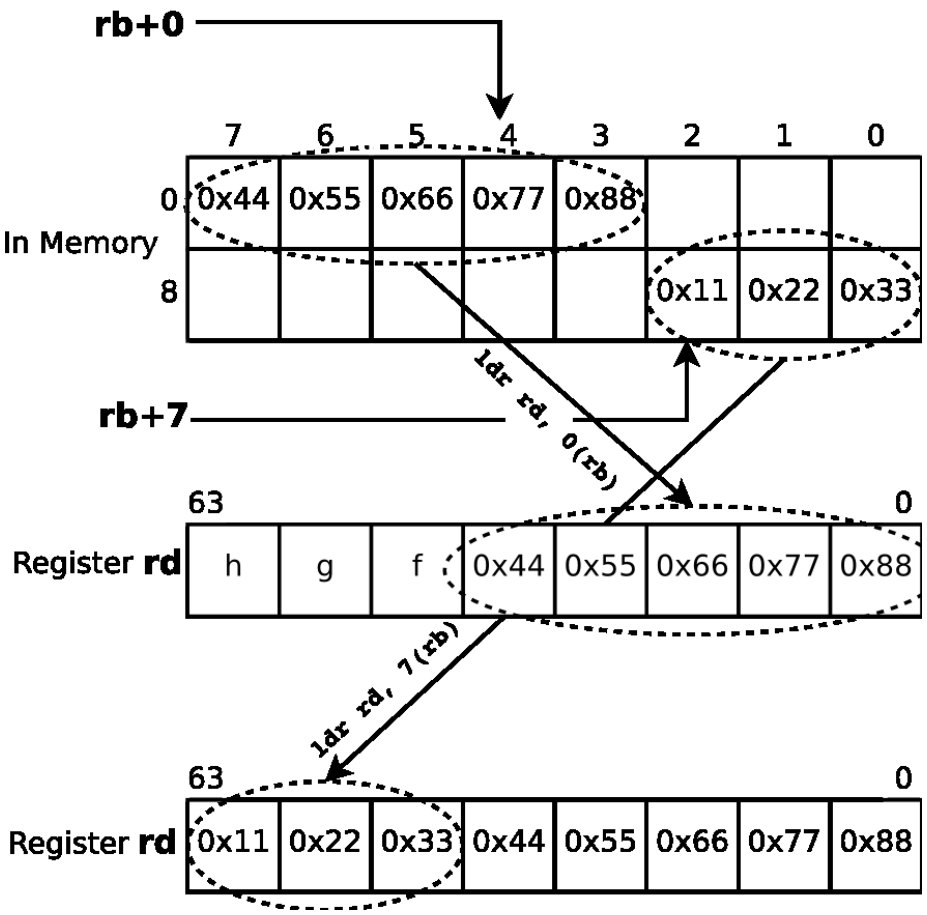

图 8.2: 在小尾端的 CPU 上加载未对齐的双字

load/store left/right 指令不要求内存控制器提供字内任意字节组的选择操作；活动的字节通道总是连续出现在字或者双字的一端。

注意这几个指令并不执行所有可能的重新对齐；对于未对齐的半字加载/存储没有专门的支持（不得不通过字节加载、移位、掩码和合并来实现）。

8.5.2 连锁加载/条件存储

指令 **ll** (load linked) 和 **sc** (store conditional) 提供了相对于作为大多数传统指令集一部分的原子性测试且设置的操作序列的替代方案。这两条指令提供了一个测试-设置序列，在运行时不保证原子性，但是仅当结果恰好是原子性执行的时候才能成功（并在成功时告知）。那样看上去更复杂，但是 **ll/sc** 在日益增大的带有相对“遥远的”的共享内存的多处理器上伸缩性很好（而原子操作伸缩性很差）。

参见第 5.8.4 节看看用于什么地方以及怎么使用。这里讲一下其工作机制。指令 **ll d, o(b)** 从常规的基址+偏移量地址执行 32 位的加载。但是它带有一个副作用，就是记住发生了一次连锁加载（在 CPU 里头设置了一个不可见的连锁状态位）。同时把加载的地址保存在寄存器 **LLAddr** 里头。

随后的 **sc, t, o(b)** 首先检查是否可以保证从上次执行的 ll 开始以来的读-改-写序列能够原子性地完成。如果能，那么 **t** 的值被存入相应的位置，真值 1 返回到 t 。如果不能保证操作的原子性，那么不执行存储操作，**t** 被设置为 0。

注意!	对于原子性的测试不可能做到彻底无遗。当 CPU 检测到某些条件表明位于相应存储器位置的值可能已经被改变但其实并没有改变的时候，指令也会失败。

有两个原因可以导致 **sc** 失败。第一个就是 CPU 在 **ll** 和 **sc** 之间执行时发生了异常。异常处理程序或者异常触发的任务切换，有可能做了某些非原子性的事情。

第二种类型的失败仅在多处理器时才会发生，当另一个 CPU 写入了该存储器位置或者接近的位置时（常见的是位于同一个高速缓存行的位置，但某些实现可能监控整个内存转换页）。由于效率原因，这个监测只有当参与的双方 CPU 都同意把该数据映射为共享数据区的时候才使能——严格来说，如果另一个 CPU 对敏感区域完成了一次连贯的存储操作。

让我们再次强调一下：**sc** 操作的失败并不能证明其它任务或 CPU 真地改动过这个变量，只是说有可能改动过，你得检查一下；鼓励具体实现时在一定数量的错报和简单性或性能之间进行适当权衡。

多处理器 CPU 必须跟踪上次 **ll** 使用的地址，把它保存在协处理器 0 的寄存器 **LLAddr** 中，在那里软件可以读写它。但是唯一读写该寄存器的原因是进行诊断；在一个没有多处理器特征的 CPU 上，这是冗余的。建议你不要依赖它。

这里给出一个简短的例子：一个对应于 Linux 内核调用 `atomic_inc(&mycount)` 的“原子”加一子程序。并发的线程——可能位于不同的 CPU 上——可以调用这个例子，每次调用把参数的值增加 1：

```
atomic_inc:
    ll      v0, 0(a0)               # a0 has pointer to 'mycount'
    addu    v0, 1
    sc      v0, 0(a0)
    beq     v0, zero, atomic_inc    # retry if sc fails
    nop
    jr      ra
    nop
```

NEC 从自己的 Vr41xx CPU 家族中略去了 **ll/sc** 指令，大概是未能意识到单处理器也能从这两条指令中受益。

8.5.3 条件传送指令

条件传送指令把数据从一个寄存器复制到另一个寄存器，但是只有在某些条件满足时才执行传送——否则，什么也不做。MIPS 的条件传送第一次出现在 MIPS IV 指令集（最先实现在 R8000、R10000 和 1995–1996 年间的 R5000 中），早在此之前，这已经成为其它的 RISC 体系结构（ARM 可能是第一个这样做的）的特色了。条件传送允许编译器生成更少的条件分支代码——这点很好，因为条件分支对于流水线效率不利。

具有象第一章描述的简单的五级流水线的 CPU，分支问题不大；分支延迟槽中的指令通常都会执行，CPU 然后直接转到分支的目标。在这些简单的 CPU 中，（只要延迟槽中包含一个有用的指令）大多数分支是免费的，其余的只花费一个时钟周期。

但是对 MIPS ISA 更复杂尖端的实现在等待确定分支条件和取出目标指令的时候可能会损失多条指令执行的机会。以 R4400 的长流水线为例，每次执行分支都要花费两个时钟周期的代价。在高度超标量的 R10000（每个周期可以发送四条指令）上，等待分支条件确定的时候也许要损失七条指令的发射机会。为了降低这种情况的影响，R10000 有个特殊的分支预测电路来猜测分支结果并提前运行，同时保持能够从这些猜测执行的指令回溯的能力。这样做非常复杂：如果编译器能够减少对复杂特性的依赖程度，程序会跑得更快。

条件传送指令是怎样去掉分支的呢？考虑计算两个值中最小值的代码片断：

```
n = (a < b) ? a : b;
```

假定编译器已经设法把所有的变量都加载进了寄存器，正常情况下编译的结果如下面的序列（这是在对延迟槽进行流水线调整之前的逻辑汇编语言序列）：

```
    slt t0, a, b
    move n, a
    bne zero, t0, 1f
    move n, b
1:
```

可以被替换为：

```
    slt t0, a, b
    move n, a
    movz n, b, t0
```

尽管条件传送指令 **movz** 看上去很奇怪，它在流水线中的作用跟别的寄存器间计算指令完全相同。去掉了分支指令，我们高度流水线化的 CPU 会跑得更快。

8.5.4 可能分支

另一个流水线优化，就是在 MIPS II 中引入的可能分支。

注意这些指令在特别复杂的流水线上很难有效地实现，所以 MIPS32 规范作废了其在可移植代码中的使用。

编译器通常能够成功填充分支延迟槽，但是在短循环的末尾时很难做到。这种循环结尾处的分支执行得最频繁，所以延迟槽中的 **nop** 影响很大；但是循环体常常充满了相互依赖的代码而很难重新组织。

可能分支指令当未发生分支时撤销分支延迟槽指令的效果。通过阻止其写回阶段可以撤销一条指令——在 MIPS 中结果就好像该指令从来没有执行过。通过仅在分支发生时才执行延迟槽指令，延迟槽指令成了下次循环的一部分。

这样任何循环：

```
loop:
    first
    second
    ...
    blez t0, loop
    nop
```

都可以变换成为：

```
loop:
    first
loop2:
    second
    ...
    blez t0, loop
    first
```

这意味着我们几乎总是可以填充循环的延迟槽，大大降低了实际执行的 **nop** 数。

你可能会看到，某些制造商的文档隐含了这样一个意思，就是说，可能分支指令通过消去 **nop** 让程序变小了；这其实是个误解。从上面的例子可以看出，**nop** 一般被一条重复指令取代，所以程序大小上并没有获益。获益的是速度。

8.5.5 整数乘加和累加指令

许多多媒体算法包括基本上是对多个乘积求和的计算。在 JPEG 图像解码器一类程序的内层循环中，这种计算多得足以让 CPU 的算术单元满负荷运行。这些计算分解为一系列乘积累加操作，每个操作就象下面这样：

```
a = a + b * c;
```

尽管按照第一章讲的 RISC 原则建议，这个运算最好通过简单、分开的功能来实现，为这个运算而破例也许值得。乘法是一个多时钟周期的运算，给简单的 RISC 留下了一个调度后续的（快速）加法的问题。如果试图太早就相加，机器就会停下来等待；如果太晚才相加，则没有让关键的算术单元保持繁忙和得到充分利用。在浮点单元中，还有一个额外的好处是与每条指令相关的某些杂务可以在乘法和加法阶段之间共享。

在 MIPS32/64 定义之前，乘累加运算属于各个厂商自己的扩展。MIPS32/64 能够保证与在 IDT、Toshiba 和 QED 的 CPU 中已经实现的操作互相兼容。运算在有独立时钟的整数乘法单元中执行，所以都是乘累加操作，[2] 把结果累加到乘法单元的输出寄存器 **lo** 和 hi。让人不解的是，所有的厂商都把他们的指令叫做 **mad** 或 **madd** 而不是“**mac**。”

8.5.6 浮点乘加指令

上面说的也都适用于浮点计算，不过这里关键的应用是 3D 图形变换和大量的矩阵运算。在浮点单元中，还有额外的好处——与每条常规指令相联系的杂务（规格化和舍入）可以在乘法和加法阶段共享。

处于大多数 PowerPC 浮点单元的中心的乘加运算毫无疑问为其应得了显赫的地位，显然这对 MIPS 采纳这几条指令有影响。

浮点运算 **madd**、**msub**、**nmadd** 和 **nmsub** 是真正的四操作数乘加指令，进行如下的运算：

```
a = b + c * d;
```

它们的目标是在 SGI 工作站上的大型图形/计算密集型应用和 SGI 的超级计算机上的重量级的数值处理。但是对于连续的乘加运算，它们并不总是生成和 IEEE 754 标准要求相同的结果（因为合并的乘加指令在加法之前并不对乘积进行舍入，所以其结果的精确度过高了）。

8.5.7 多个浮点条件位

在 MIPS IV 之前，对于浮点数操作结果的所有测试和主指令集间的通信都是通过一个单个的条件位，该位由比较指令明确设置，由特殊的条件分支指令明确测试。体系结构这样的发展，是因为早期浮点单元是一个独立的芯片，浮点条件位是通过一个传递到主 CPU 的信号线来实现的。

单个位的麻烦在于导致了依赖，降低了并行发送多个指令的潜力。在创建条件的比较指令和测试条件的分支指令之间，有一个不可避免的写-读依赖；但

[2] Toshiba 的 R3900 和其它一些 CPU 有个三操作数的乘加指令，即便如此，被加数被限制只能来自 **hi/lo**。IDT 和 QED 的 CPU 提供了一条双操作数指令，跟 Toshiba 的指令在目标操作数为 $0 的特例下完全相同。

在随后的比较指令必须延迟到分支看到前一个值并按它行动的地方，则有一个可以避免的读-写依赖。

浮点数组计算受益于一种叫做软件流水线的编译技术，该技术把循环展开，将相继的几次循环的计算有意相互交错，以最大限度的利用浮点单元。但如果循环体的某部分需要测试和分支，单个条件位就不能做到这点；这时候多个条件位可以起很大作用。

现代的 FPU 提供 8 个条件位，而不是只有 1 位；比较和浮点条件分支指令可以指定用哪个条件位。老式编译器把保留域设置为 0，所以只用条件位 0 的老代码也会正确运行。

8.5.8 预取

pref 提供了一种方法，可以让程序示意高速缓存/存储器系统说，某些数据很快会用到。利用这一特性的实现可以把数据预取到高速缓存。到底有多少应用程序能够预见到哪些数据引用可能导致高速缓存未命中，这一点还不真正清楚；然而，预取对于大型的数组运算功能很有用，可以在一次循环中预取出大批的数据为下一次循环作准备。

pref 的第一个参数是一个小整数，用以编码有关程序想怎样使用数据的“提示(hint)”。一些 MIPS32/64 CPU 实现的一些取值范围如表 8.5 所示。

如果 MIPS CPU 并不能理解某个提示，可以当作“load”提示或者完全忽略。一般来说，CPU 完全可以随意忽略 **pref**，把它当成 **nop**。这样虽然采用提示的优化针对于某个特定的 CPU，但在其它的 CPU 上也不会破坏代码。

有些 MIPS CPU 实现了非阻塞的加载，其中在加载时碰到高速缓存未命中后，只要没有引用加载的目标寄存器，就向下继续执行。然而，**pref** 指令更适用于对存储器访问的长程预测（在没有实现的 CPU 上蜕化为一个空操作，比起在程序中特意提早执行的阻塞加载这种做法，明显更为良性）。

8.5.9 Sync：用于 load/store 的存储器防护

假定我们有一个程序由一些互相协作的串行任务组成，每个都运行于不同的“处理器” 上并且共享内存。我们可能在谈论采用了复杂的高速缓存一致性算法的多处理器，但是现在先不考虑高速缓存管理。当另外的“处理器”是个采用用 DMA 的 I/O 控制器的时候，也会有同样的问题。

任何任务的健壮的共享内存算法要依赖于其它任务在什么时候访问过共享数据：其它任务在我修改之前读过数据码？它们修改过数据吗？

既然每个任务是严格串行有序的，为什么这个会成为一个问题？出现这个问题是因为 CPU 的性能调优特性经常和内存操作的逻辑顺序相干扰；按照定义，该干扰对程序自身必须是不可见的，但是从外面看时就会暴露出来。有许多正当的理由可以打破自然的顺序。为了最佳的内存性能，读操作——在 CPU 因为

表 8.5: 预取提示代码

值/MIPS 名字	可能会发生什么	什么时候使用
0 - load 1 - store	如果还没有就把高速缓存行读到 D-cache	你期望很快要读取数据的时候。如果你也想修改它就用“store”提示。
4 - load streamed 5 - store streamed	避免每个数据只用一次的流式数据占满整个高速缓存。也许只用某一路高速缓存。	用于你期望顺序进行处理并且处理之后立刻可以丢弃的数据。
6 - load retained 7 - store retained	与流式相反（所以也许是要用其它路高速缓存）。	用于期望多次使用并可能和流式数据竞争高速缓存的数据。
25 - writeback_invalidate/nudge	如果该行位于高速缓存行内，就作废（如果被改过要先回写）。	当你知道你已经处理完了数据，想要保证其未来对高速缓存资源的竞争必然失败时。
30 - PrepareForStore	若相应行并未进入高速缓存，创建相应的高速缓存行——但不是从存储器读取数据，而是将其用零填充并且标记为脏。若该行已经位于高速缓存内，什么也不做——不能依赖该操作对高速缓存行清零。	当你知道你将要重写整个行的时候，从内存读取老的数据就没有必要了。一个回收的行要用零填充只是因为其以前的内容可能属于某个敏感的应用——让新的程序看到以前的数据可能会破坏安全性。

等待数据而停顿时——应当越过正在等待的写操作。只要 CPU 在一次写的时候同时存储数据和地址，就可以把写操作推迟一会儿。如果 CPU 那样做，最好检查一下确认读取的地址不是等待写的位置；这点可以做到。

另一个例子是当实现了非阻塞加载的 CPU 最后碰到同时有两个读操作在活动；为了达到最佳性能，应当允许存储器系统选择先执行哪个。

不允许以上任何一种顺序改变，严格按照程序顺序执行所有的读写操作的 CPU，称做强有序*(strongly ordered)*的。许多 MIPS CPU 配制成单处理器的时候，都是强有序的。但也有例外：即使有些早期的 R3000 系统也允许读操作越过等待的写（要先检查读的不是任何等待写的位置）。

一条 **sync** 指令定义一个存取防护。可以向你保证所有在 **sync** 之前发起的存取的结果，位于 **sync** 后面的任何存取操作都可见到。

注意在一个多处理器中，我们不得不坚持“可见到”这个词是指“系统中任何正确实现了共享存储高速缓存系统的任务都可见到”。通常的做法是保证 **sync** 为 CPU 和高速缓存/内存/总线子系统之间的事务生成一个防止重新排序的屏障。

也有一些限制。对于存取操作和 **sync** 本身执行的相对时序没有保证；仅仅是把该指令之前和之后的存取操作分开。**sync** 不能保证解决 CPU 的程序执行和外部写之间的时序关系问题，这个问题我们在 10.4 节会提到。

在一个系统里头，**sync** 只在某些访问类型（不用高速缓存的访问以及一致高速缓存的访问）上起作用。很多“正常”的经过高速缓存的存储器不是一致的，任何已知不会被共享的数据空间都是安全的，在共享任务之间只读的数据也一样。

在强有序的 CPU 上，**sync** 什么都不用做；这种情形，可以就是一条 **nop**。

然而，通常 **sync** 都有些矫枉过正；请查阅你的 CPU 手册。

8.5.10 遇险防护指令

一条指令使用紧挨在前面的指令的结果再也平常不过了。在流水线化的处理器中，正常第二条指令从寄存器堆读取操作数的时间在前一条的值写回之前。那不是问题，因为 MIPS 指令仅仅在寄存器中传递数据，进出通用寄存器和浮点寄存器的数据大量是经过旁路的——就是说，硬件监测这种依赖并且安排将数据直接前推到第二个指令正好需要的时候。

但是对于 CP0 寄存器（一般）不是这样做的。如果你写一个 CP0 寄存器的域，这也许会影响随后的指令，MIPS 的规定并不保证这个影响要花多久。

有两种类型的遇险。最明显的就是依赖的指令要用第一个指令提供的值：那叫做*执行遇险*。

更麻烦的情形是写操作改变了某些 CPU 的状态，以至于甚至会影响到随后的指令的取指：比如说，当对 CP0 的写修改了存储器映射转换的设置时，就会出现这种情况。这些叫做*指令遇险*。

传统上，MIPS CPU 把设计能够保证正确运行的序列的工作留给了内核/底层软件工程师，一般通过在写操作和依赖指令之间加上足够数量的 **nop** 或者 **ssnop** [3]指令来实现。

自 MIPS32 第二版规范发行以来，这种做法已经被*遇险防护指令*所取代。**eret**、**jr.hb** 和 **jalr** 能防护包括执行遇险在内的所有副作用，而 **ehb**（一种增强的 no-op）以较小的开销处理执行遇险。

有关遇险和防护更多的讨论，参见第 3.4 节。

移植软件来使用新指令

如果你知道你的软件只在 MIPS32 第二版或更高版本的 CPU 上运行，那么就太好了。但是为了维护不得不继续运行在老的 CPU 上的软件，新的版规范保证：

- *ehb 等于 no-op*: 在以前所有的 CPU 上都成立。你可以创建出一种软件，既能继续运行于老式 CPU 上，又在未来的所有 MIPS32/64 兼容的 CPU

[3]**ssnop** 是一个特殊类型的 no-op，能够保证在那种一个时钟周期可以发射多个指令的 CPU 上一条指令自身就占用一整个发射周期。

上仍然没问题：只要用 ehb 来替换你的“足够多的 no-op”中的最后一个就行了。

- *jr.hb* 和 *jalr.hb*: 在早期的 CPU 上译码为简单的寄存器跳转和寄存器调用指令。只要已经在你的最老的 CPU 上有了足够多的 no-op，现在你的系统在 MIPS32/64 第二版的 CPU 上也就是安全的。

8.5.11 synci: 为改写指令的程序做高速缓存管理

把另一个程序加载进内存的程序实际上写入的是数据方的高速缓存。装载的指令要进入指令高速缓存后才能执行。MIPS CPU 没有连接两个高速缓存的逻辑（两个高速缓存都是性能相关的，所以为此而增加逻辑没有太大的意义，因为很少用到）。

在指令被写入后，加载程序应当安排回写任何包含的数据高速缓存行，并且作废已经位于指令高速缓存相应位置的内容。你当然可以用第 4.6 节讲的 **cache** 指令做到这一点——但这些指令只能在核心态使用，一个为供自身进程使用而写些指令的装载程序不一定得是特权软件。

因此，在最新的 MIPS32/64 CPU 中，MIPS 提供了 **synci** 指令，为刚刚加载的相当于高速缓存行大小的内存块完成整个工作：就是说，既安排数据高速缓存 D-cache 回写，又安排指令高速缓存 I-cache 作废。

要在用户级使用 **synci**，你需要知道高速缓存行的大小，这可以通过一条指令 **rdhwr SYNCI_Step** 从下节讲的一个标准“硬件寄存器”得到。

8.5.12 读取硬件寄存器

rdhwr 直接给非特权的（用户态）软件提供了关于硬件的有用信息。

MIPS32/64 规范目前定义了四个寄存器。操作系统通过在 CP0 寄存器 **HWREna** 设置位屏蔽（位 0 置位使能寄存器 0，等等。）可以单独控制对每个寄存器的访问。**HWREna** 在复位时全部清零，所以必须用软件明确使能用户访问。特权代码永远可以读取一切信息，无视 HWREna 中的零。

四个寄存器为：

- CPUNum(0): 该程序当前正在运行的 CPU 号。这可以从协处理器 0 的 **EBase(CPUNum)** 域得来。

- SYNCI_Step(1): 一级高速缓存行的有效宽度。[4]

  行的宽度对于用户程序很重要，因为他们现在可以用 **synci** 指令对高速缓存操作使得写入的指令可以执行。然后 **SYNCI_Step** 告诉你“步长”——要覆盖某个范围内的全部指令而要求的相继 **synci** 之间的地址增量。

[4] 严格来说，是指令高速缓存 I-cache 和数据高速缓存 D-cache 行中较小的，但是二者不一样的情形极为罕见。

如果 **SYNCI_Step** 返回零值，意思就是说你根本不需要用 **synci**。

- CC(2): 用户态对 CP0 **Count** 寄存器只读访问，用于高分辨率计数。这个没有太大用处，除非...
- CCRes(3): 告诉你 **Count** 计数有多快。它是对流水线时钟的分频因子（如果你读到的值为 "2"，那么 Count 以流水线时钟频率的一半，每两个周期增加一次）。

8.6 指令编码

MIPS32/64 ISA 第二版定义的全部 MIPS 指令（以及经过慎重选择的由各种体系结构变体所定义的不同指令）按照编码顺序列在表 8.6 中。第 8.6.2 节和第 8.6.3 节提供了关于表中资料进一步的信息。

大多数的 MIPS 手册说只有三种指令格式使用过。我敢说这对应于芯片最初内部设计的一些现实，但对用户看来绝对不象那样，在用户看来好像是不同的指令将各个域用于完全不同的目的。新的指令用了更复杂的编码。

表 8.6 告诉你二进制编码和汇编代码中的指令助记符。

标题为"所属 ISA"的列标出没有被所有 MIPS32 兼容的 CPU 实现的指令：该列的内容包括"R2，"表示直到 MIPS32/64 第二版才要求的指令，"MIPS64"表示只有 64 位 CPU 才必须的指令，"EJTAG"表示与调试单元相联系的指令，"3D"和"PS"表示与浮点可选的扩展 MIPS-3D 和单精度浮点对相关的，"非 MIPS32/64"表示曾经用于 MIPS I 但现在已经过时不用了。偶尔最后一列会有某个提供特殊指令的具体 CPU 名字。

8.6.1 指令编码表中的各个域

下面的内容解释一些表 8.6 中的域。

Field 31–26 主操作码"op"，为 6 位长度。难以放进 32 位的指令（象"长程"的 j 和 jal 指令或带 16 位常数的算术指令）有个唯一的"op" 域。其它指令以组为单位共享一个"op"值，通过别的域区分。

Field 5–0 用于三寄存器操作数的算术/逻辑指令组的子码域（主操作码为零）。

Field 25–21 又一个扩展的操作码域，这次由协处理器类型的指令使用。

s, t, w 标识源寄存器的域，偶尔当不确定是通用寄存器的时候用 rs, rt。

o(b), offset "o"是放进 16 位域的一个有符号偏移量；通用的基址寄存器，其内容被加到"o"上生成一个给 load 或 store 指令用的地址。

d 目标寄存器，要被该指令修改的。偶尔我们写成 rd 以提醒你是一个通用寄存器。

shf 移位量的多少，用在常数移位指令中。

broffset 一个有符号的 16 位相对于 PC 的字偏移量，代表以字为单位的到某个标号的距离（每个指令一个字）。零偏移量代表分支后的延迟槽指令，所以分支到自己的偏移量是 -1。

target 要跳去的 26 位的字地址（对应于 28 位的字节地址）。长程跳转指令 **j** 很少使用，所以这个格式几乎完全用于函数调用（**jal**）。

目标地址的高 4 位无法用本指令给出，从跳转指令的地址得到。这意味着这些指令可以到达在该指令的地址附近的 256 M 范围之内。要跳到更远的地方，用 **jr** （寄存器跳转）指令。

constant 用于算术逻辑操作的 16 位的整型常数直接量。根据不同指令可以解释为有符号或者无符号数。

cs/cd 作为来源或目标的协处理器寄存器。指令集的每个协处理器部分可以有最多 32 个数据寄存器和 32 个控制寄存器。

fr/fs/ft 浮点单元源寄存器。

fd 浮点目标寄存器（由指令写入数据）。

N/M 浮点条件码选择器——当读取时为“N”，由比较指令写入时为“M”。在只有一个浮点条件码的老式浮点指令集中该域为零，因为老的汇编代码没有这个域，你写指令时可以不写该域，等效于第零个条件码。

hint 8.5.8 节介绍的预取指令提示。

cachop 这个和 **cache** 指令一起使用，对在指令地址中发现的高速缓存项的操作进行编码。参见第 4.9 节的表 4.2。

表 8.6: 按照编码排序的机器指令

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
0	0	0		0	0		0	**nop**	
0	0	0		0	1		0	**ssnop**	
0	0	w		d	shf		0	**sll d,w,shf**	
0	s	N	0	d	0		1	**movf d,s,N**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
0	s	N	1	d	0		1	**movt d,s,N**	
0	0	w		d	shf		2	**srl d,w,shf**	
0	0	w		d	shf		2	**rotr d,w,shf**	R2
0	0	w		d	shf		3	**sra d,w,shf**	
0	s	t		d	0		4	**sllv d,t,s**	
0	s	t		d	0		6	**srl d,t,s**	
0	s	t		d	1		6	**rotrv d,t,s**	R2
0	s	t		d	0		7	**srav d,t,s**	
0	s	0		0	0		8	**jr s**	
0	s	0		0	16		8	**jr.hb s**	
0	s	0		31	0		9	**jalr s**	
0	s	0		d	0		9	**jalr d,s**	
0	s	0		d	16		9	**jalr.hb d,s**	
0	s	t		d	0		10	**movz d,s,t**	
0	s	t		d	0		11	**movn d,s,t**	
0	code						12	**syscall code**	
0	code			x			13	**break code**	
0	code			x			14	**sdbbp code**	R3900
0	0	0		0	0		15	**sync**	
0	0	0		d	0		16	**mfhi d**	
0	s	0		0	0		17	**mthi s**	
0	0	0		d	0		18	**mflo d**	
0	s	0		0	0		19	**mtlo s**	
0	s	t		d	0		20	**dsllv d,t,s**	MIP64
0	s	t		d	0		22	**dsrlv d,t,s**	MIP64
0	s	t		d	1		22	**drotrv d,t,s**	MIP64R2
0	s	t		d	0		23	**dsrav d,t,s**	MIP64
0	s	t		0	0		24	**mult s,t**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
0	s	t		0	0		25	**multu s,t**	
0	s	t		0	0		26	**div s,t**	
0	s	t		0	0		27	**divu s,t**	
0	s	t		0	0		28	**dmult s,t**	MIPS64
0	s	t		0	0		29	**dmultu s,t**	MIPS64
0	s	t		0	0		30	**ddiv s,t**	MIPS64
0	s	t		0	0		31	**ddivu s,t**	MIPS64
0	s	t		d	0		32	**add d,s,t**	
0	s	t		d	0		33	**addu d,s,t**	
0	s	t		d	0		34	**sub d,s,t**	
0	s	t		d	0		35	**subu d,s,t**	
0	s	t		d	0		36	**and d,s,t**	
0	s	t		d	0		37	**or d,s,t**	
0	s	t		d	0		38	**xor d,s,t**	
0	s	t		d	0		39	**nor d,s,t**	
0	s	t		0	0		40	**madd16 d,s,t**	Vr4100
0	s	t		0	0		41	**dmadd16 d,s,t**	Vr4100
0	s	t		d	0		42	**slt d,s,t**	
0	s	t		d	0		43	**sltu d,s,t**	
0	s	t		d	0		44	**dadd d,s,t**	MIPS64
0	s	t		d	0		45	**daddu d,s,t**	MIPS64
0	s	t		d	0		46	**dsub d,s,t**	MIPS64
0	s	t		d	0		47	**dsubu d,s,t**	MIPS64
0	s	t		x			48	**tge s,t**	
0	s	t		x			49	**tgeu s,t**	
0	s	t		x			50	**tlt s,t**	
0	s	t		x			51	**tltu s,t**	
0	s	t		x			52	**teq s,t**	
0	s	t		x			54	**tne s,t**	
0	0	w		d	shf		56	**dsll d,w,shf**	MIP64
0	0	w		d	shf		58	**dsrl d,w,shf**	MIP64
1	0	w		d	shf		58	**drotr d,w,shf**	MIP64R2
0	0	w		d	shf		59	**dsra d,w,shf**	MIP64

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
0	0	w		d	shf		60	**dsll32 d,w,shf**	MIP64
0	0	w		d	shf		62	**dsrl32 d,w,shf**	MIP64
0	0	w		d	shf		63	**dsra32 d,w,shf**	MIP64
1	s	0		broffset				**bltz s,p**	
1	s	1		broffset				**bgez s,p**	
1	s	2		broffset				**bltzl s,p**	
1	s	3		broffset				**bgezl s,p**	
1	s	8		constant				**tgei s,j**	
1	s	9		constant				**tgeiu s,j**	
1	s	10		constant				**tlti s,j**	
1	s	11		constant				**tltiu s,j**	
1	s	12		constant				**teqi s,j**	
1	s	14		constant				**tnei s,j**	
1	s	16		broffset				**bltzal s,p**	
1	s	17		broffset				**bgezal s,p**	
1	s	18		broffset				**bltzall s,p**	
1	s	19		broffset				**bgezall s,p**	
1	b	31		o				**synci o(b)**	R2
2	target							**j target**	
3	target							**jal target**	
4	s	t		broffset				**beq s,t,p**	
5	s	t		broffset				**bne s,t,p**	
6	s	0		broffset				**blez s,p**	
7	s	0		broffset				**bgtz s,p**	
8	s	d		(signed) const				**addi d,s,const**	
9	s	d		(signed) const				**addiu d,s,const**	
10	s	d		(signed) const				**slti d,s,const**	
11	s	d		(signed) const				**sltiu d,s,const**	
12	s	d		(unsigned) const				**andi d,s,const**	
13	s	d		(unsigned) const				**ori d,s,const**	
14	s	d		(unsigned) const				**xori d,s,const**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
15	0	d		(unsigned) const				**lui d,s,const**	
16	0	t		cs	0		0	**mfc0 t,cs**	
16	1	t		cs	0		0	**dmfc0 t,cs**	MIP64
16	2	t		cs	0		0	**cfc0 t,cs**	MIP64
16	4	t		cd	0		0	**mtc0 t,cd**	
16	5	t		cd	0		0	**dmtc0 t,cs**	MIP64
16	10	xt		d	0		0	**rdpgpr d,xt**	R2
16	11	t		12	0		0	**di t**	R2
16	11	t		12	0		32	**ei t**	R2
16	14	t		xd	0		0	**wrpgpr xd,t**	R2
16	16	0		0	0		1	**tlbr**	
16	16	0		0	0		2	**tlbwi**	
16	16	0		0	0		6	**tlbwr**	
16	16	0		0	0		8	**tlbp**	
16	16	0		0	0		16	**rfe**	MIPS I
16	16	0		0	0		24	**eret**	
16	16	0		0	0		31	**dret**	MIPS II
16	16	0		0	0		32	**deret**	EJTAG
16	16	0		0	0		33	**standby**	Vr4100
16	16	0		0	0		34	**suspend**	Vr4100
16	8	0		broffset				**bc0f p**	非 MIPS32/64
16	8	1		broffset				**bc0t p**	非 MIPS32/64
16	8	2		broffset				**bc0fl p**	非 MIPS32/64
16	8	3		broffset				**bc0tl p**	非 MIPS32/64
17	0	t		fs	0		0	**mfc1 t,fs**	
17	1	t		fs	0		0	**dmfc1 t,fs**	MIPS64
17	2	t		cs	0		0	**cfc1 t,cs**	
17	3	t		fs	0		0	**mfhc1 t,fs**	R2

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	4	t		cs	0		0	**mtc1 t,fs**	
17	5	t		cs	0		0	**dmtc1 t,fs**	MIPS64
17	6	t		cs	0		0	**ctc1 t,fs**	
17	7	t		fs	0		0	**mthc1 t,fs**	R2
17	8	N	0	broffset				**bc1f N,p**	
17	8	N	1	broffset				**bc1t N,p**	
17	8	N	2	broffset				**bc1fl N,p**	
17	8	N	3	broffset				**bc1tl N,p**	
17	9	N	0	broffset				**bc1any2f N,p**	3D
17	9	N	1	broffset				**bc1any2t N,p**	3D
17	9	N	2	broffset				**bc1any4f N,p**	3D
17	9	N	3	broffset				**bc1any4t N,p**	3D
17	16	ft		fs	fd		0	**add.s fd,fs,ft**	
17	17	ft		fs	fd		0	**add.d fd,fs,ft**	
17	22	ft		fs	fd		0	**add.ps fd,fs,ft**	PS
17	16	ft		fs	fd		1	**sub.s fd,fs,ft**	
17	17	ft		fs	fd		1	**sub.d fd,fs,ft**	
17	22	ft		fs	fd		1	**sub.ps fd,fs,ft**	PS
17	16	ft		fs	fd		2	**mul.s fd,fs,ft**	
17	17	ft		fs	fd		2	**mul.d fd,fs,ft**	
17	22	ft		fs	fd		2	**mul.ps fd,fs,ft**	PS
17	16	ft		fs	fd		3	**div.s fd,fs,ft**	
17	17	ft		fs	fd		3	**div.d fd,fs,ft**	
17	16	0		fs	fd		4	**sqrt.s fd,fs**	
17	17	0		fs	fd		4	**sqrt.d fd,fs**	
17	16	0		fs	fd		5	**abs.s fd,fs**	
17	17	0		fs	fd		5	**abs.d fd,fs**	
17	22	0		fs	fd		5	**abs.ps fd,fs**	PS
17	16	0		fs	fd		6	**mov.s fd,fs**	
17	17	0		fs	fd		6	**mov.d fd,fs**	
17	22	0		fs	fd		6	**mov.ps fd,fs**	PS
17	16	0		fs	fd		7	**neg.s fd,fs**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	17	0		fs	fd		7	**neg.d fd,fs**	
17	22	0		fs	fd		7	**neg.ps fd,fs**	PS
17	16	0		fs	fd		8	**round.l.s fd,fs**	MIPS64
17	17	0		fs	fd		8	**round.l.d fd,fs**	MIPS64
17	16	0		fs	fd		9	**trunc.l.s fd,fs**	MIPS64
17	17	0		fs	fd		9	**trunc.l.d fd,fs**	MIPS64
17	16	0		fs	fd		10	**ceil.l.s fd,fs**	MIPS64
17	17	0		fs	fd		10	**ceil.l.d fd,fs**	MIPS64
17	16	0		fs	fd		11	**floor.l.s fd,fs**	MIPS64
17	17	0		fs	fd		11	**floor.l.d fd,fs**	MIPS64
17	16	0		fs	fd		12	**round.w.s fd,fs**	
17	17	0		fs	fd		12	**round.w.d fd,fs**	
17	16	0		fs	fd		13	**trunc.w.s fd,fs**	
17	17	0		fs	fd		13	**trunc.w.d fd,fs**	
17	16	0		fs	fd		14	**ceil.w.s fd,fs**	
17	17	0		fs	fd		14	**ceil.w.d fd,fs**	
17	16	0		fs	fd		15	**floor.w.s fd,fs**	
17	17	0		fs	fd		15	**floor.w.d fd,fs**	
17	16	N	0	fs	fd		17	**movf.s fd,fs,N**	
17	17	N	0	fs	fd		17	**movf.d fd,fs,N**	
17	22	N	0	fs	fd		17	**movf.ps fd,fs,N**	PS
17	16	N	1	fs	fd		17	**movt.s fd,fs,N**	
17	17	N	1	fs	fd		17	**movt.d fd,fs,N**	
17	16	t		fs	fd		18	**movz.s fd,fs,t**	
17	17	t		fs	fd		18	**movz.d fd,fs,t**	
17	22	t		fs	fd		18	**movz.ps fd,fs,t**	PS
17	16	t		fs	fd		19	**movn.s fd,fs,t**	
17	17	t		fs	fd		19	**movn.d fd,fs,t**	
17	22	t		fs	fd		19	**movn.ps fd,fs,t**	PS
17	16	0		fs	fd		21	**recip.s fd,fs**	
17	17	0		fs	fd		21	**recip.d fd,fs**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	16	0		fs	fd		22	**rsqrt.s fd,fs**	
17	17	0		fs	fd		22	**rsqrt.d fd,fs**	
17	22	0		fs	fd		24	**addr.ps fd,fs**	3D
17	22	0		fs	fd		26	**mulr.ps fd,fs**	3D
17	17	0		fs	fd		28	**recip2.d fd,fs**	3D
17	22	0		fs	fd		28	**recip2.ps fd,fs**	3D
17	17	0		fs	fd		29	**recip1.d fd,fs**	3D
17	22	0		fs	fd		29	**recip1.ps fd,fs**	3D
17	17	0		fs	fd		30	**rsqrt1.d fd,fs**	3D
17	22	0		fs	fd		30	**rsqrt1.ps fd,fs**	3D
17	17	0		fs	fd		31	**rsqrt2.d fd,fs**	3D
17	22	0		fs	fd		31	**rsqrt2.ps fd,fs**	3D
17	17	0		fs	fd		32	**cvt.s.d fd,fs**	
17	20	0		fs	fd		32	**cvt.s.w fd,fs**	
17	21	0		fs	fd		32	**cvt.s.l fd,fs**	MIPS64
17	22	0		fs	fd		32	**cvt.s.pu fd,fs**	PS
17	16	0		fs	fd		33	**cvt.d.s fd,fs**	
17	20	0		fs	fd		33	**cvt.d.w fd,fs**	
17	21	0		fs	fd		33	**cvt.d.l fd,fs**	MIPS64
17	16	0		fs	fd		36	**cvt.w.s fd,fs**	
17	17	0		fs	fd		36	**cvt.w.d fd,fs**	
17	22	0		fs	fd		36	**cvt.pw.ps fd,fs**	3D
17	16	0		fs	fd		37	**cvt.l.s fd,fs**	MIPS64
17	17	0		fs	fd		37	**cvt.l.d fd,fs**	MIPS64
17	16	0		fs	fd		38	**cvt.ps.s fd,fs**	PS
17	20	0		fs	fd		38	**cvt.ps.pw fd,fs**	3D
17	21	0		fs	fd		38	**cvt.ps.pw.l fd,fs**	PS
17	22	0		fs	fd		40	**cvt.s.pl fd,fs**	PS
17	22	0		fs	fd		44	**pll.ps.ps fd,fs**	PS
17	22	0		fs	fd		45	**plu.ps.ps fd,fs**	PS
17	22	0		fs	fd		46	**pul.ps.ps fd,fs**	PS
17	22	0		fs	fd		47	**puu.ps.ps fd,fs**	PS

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	16	ft		fs	M	0	48	**c.f.s M,fs,ft**	
17	17	ft		fs	M	0	48	**c.f.d M,fs,ft**	
17	22	ft		fs	M	0	48	**c.f.ps M,fs,ft**	PS
17	16	ft		fs	M	0	49	**c.un.s M,fs,ft**	
17	17	ft		fs	M	0	49	**c.un.d M,fs,ft**	
17	22	ft		fs	M	0	49	**c.un.ps M,fs,ft**	PS
17	16	ft		fs	M	0	50	**c.eq.s M,fs,ft**	
17	17	ft		fs	M	0	50	**c.eq.d M,fs,ft**	
17	22	ft		fs	M	0	50	**c.eq.ps M,fs,ft**	PS
17	16	ft		fs	M	0	51	**c.ueq.s M,fs,ft**	
17	17	ft		fs	M	0	51	**c.ueq.d M,fs,ft**	
17	22	ft		fs	M	0	51	**c.ueq.ps M,fs,ft**	PS
17	16	ft		fs	M	0	52	**c.olt.s M,fs,ft**	
17	17	ft		fs	M	0	52	**c.olt.d M,fs,ft**	
17	22	ft		fs	M	0	52	**c.olt.ps M,fs,ft**	PS
17	16	ft		fs	M	0	53	**c.ult.s M,fs,ft**	
17	17	ft		fs	M	0	53	**c.ult.d M,fs,ft**	
17	22	ft		fs	M	0	53	**c.ult.ps M,fs,ft**	PS
17	16	ft		fs	M	0	54	**c.ole.s M,fs,ft**	
17	17	ft		fs	M	0	54	**c.ole.d M,fs,ft**	
17	22	ft		fs	M	0	54	**c.ole.ps M,fs,ft**	PS
17	16	ft		fs	M	0	55	**c.ule.s M,fs,ft**	
17	17	ft		fs	M	0	55	**c.ule.d M,fs,ft**	
17	22	ft		fs	M	0	55	**c.ule.ps M,fs,ft**	PS
17	16	ft		fs	M	0	56	**c.sf.s M,fs,ft**	
17	17	ft		fs	M	0	56	**c.sf.d M,fs,ft**	
17	22	ft		fs	M	0	56	**c.sf.ps M,fs,ft**	PS
17	16	ft		fs	M	0	57	**c.ngle.s M,fs,ft**	PS
17	17	ft		fs	M	0	57	**c.ngle.d M,fs,ft**	PS
17	22	ft		fs	M	0	57	**c.ngle.ps M,fs,ft**	PS
17	16	ft		fs	M	0	58	**c.seq.s M,fs,ft**	
17	17	ft		fs	M	0	58	**c.seq.d M,fs,ft**	
17	22	ft		fs	M	0	58	**c.seq.ps M,fs,ft**	PS

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	16	ft		fs	M	0	59	**c.ngl.s M,fs,ft**	
17	17	ft		fs	M	0	59	**c.ngl.d M,fs,ft**	
17	22	ft		fs	M	0	59	**c.ngl.ps M,fs,ft**	PS
17	16	ft		fs	M	0	60	**c.lt.s M,fs,ft**	
17	17	ft		fs	M	0	60	**c.lt.d M,fs,ft**	
17	22	ft		fs	M	0	60	**c.lt.ps M,fs,ft**	PS
17	16	ft		fs	M	0	61	**c.nge.s M,fs,ft**	
17	17	ft		fs	M	0	61	**c.nge.d M,fs,ft**	
17	22	ft		fs	M	0	61	**c.nge.ps M,fs,ft**	PS
17	16	ft		fs	M	0	62	**c.le.s M,fs,ft**	
17	17	ft		fs	M	0	62	**c.le.d M,fs,ft**	
17	22	ft		fs	M	0	62	**c.le.ps M,fs,ft**	PS
17	16	ft		fs	M	0	63	**c.ngt.s M,fs,ft**	
17	17	ft		fs	M	0	63	**c.ngt.d M,fs,ft**	
17	22	ft		fs	M	0	63	**c.ngt.ps M,fs,ft**	PS
17	16	ft		fs	M	1	48	**cabs.f.s M,fs,ft**	3D
17	17	ft		fs	M	1	48	**cabs.f.d M,fs,ft**	3D
17	22	ft		fs	M	1	48	**cabs.f.ps M,fs,ft**	3D
17	16	ft		fs	M	1	49	**cabs.un.s M,fs,ft**	3D
17	17	ft		fs	M	1	49	**cabs.un.d M,fs,ft**	3D
17	22	ft		fs	M	1	49	**cabs.un.ps M,fs,ft**	3D
17	16	ft		fs	M	1	50	**cabs.eq.s M,fs,ft**	3D
17	17	ft		fs	M	1	50	**cabs.eq.d M,fs,ft**	3D
17	22	ft		fs	M	1	50	**cabs.eq.ps M,fs,ft**	3D
17	16	ft		fs	M	1	51	**cabs.ueq.s M,fs,ft**	3D
17	17	ft		fs	M	1	51	**cabs.ueq.d M,fs,ft**	3D
17	22	ft		fs	M	1	51	**cabs.ueq.ps M,fs,ft**	3D
17	16	ft		fs	M	1	52	**cabs.olt.s M,fs,ft**	3D
17	17	ft		fs	M	1	52	**cabs.olt.d M,fs,ft**	3D
17	22	ft		fs	M	1	52	**cabs.olt.ps M,fs,ft**	3D
17	16	ft		fs	M	1	53	**cabs.ult.s M,fs,ft**	3D
17	17	ft		fs	M	1	53	**cabs.ult.d M,fs,ft**	3D
17	22	ft		fs	M	1	53	**cabs.ult.ps M,fs,ft**	3D
17	16	ft		fs	M	1	54	**cabs.ole.s M,fs,ft**	3D

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
17	17	ft		fs	M	1	54	**cabs.ole.d M,fs,ft**	3D
17	22	ft		fs	M	1	54	**cabs.ole.ps M,fs,ft**	3D
17	16	ft		fs	M	1	55	**cabs.ule.s M,fs,ft**	3D
17	17	ft		fs	M	1	55	**cabs.ule.d M,fs,ft**	3D
17	22	ft		fs	M	1	55	**cabs.ule.ps M,fs,ft**	3D
17	16	ft		fs	M	1	56	**cabs.sf.s M,fs,ft**	3D
17	17	ft		fs	M	1	56	**cabs.sf.d M,fs,ft**	3D
17	22	ft		fs	M	1	56	**cabs.sf.ps M,fs,ft**	3D
17	22	ft		fs	M	1	57	**cabs.ngle.ps M,fs,ft**	3D
17	16	ft		fs	M	1	58	**cabs.seq.s M,fs,ft**	3D
17	17	ft		fs	M	1	58	**cabs.seq.d M,fs,ft**	3D
17	22	ft		fs	M	1	58	**cabs.seq.ps M,fs,ft**	3D
17	16	ft		fs	M	1	59	**cabs.ngl.s M,fs,ft**	3D
17	17	ft		fs	M	1	59	**cabs.ngl.d M,fs,ft**	3D
17	22	ft		fs	M	1	59	**cabs.ngl.ps M,fs,ft**	3D
17	16	ft		fs	M	1	60	**cabs.lt.s M,fs,ft**	3D
17	17	ft		fs	M	1	60	**cabs.lt.d M,fs,ft**	3D
17	22	ft		fs	M	1	60	**cabs.lt.ps M,fs,ft**	3D
17	16	ft		fs	M	1	61	**cabs.nge.s M,fs,ft**	3D
17	17	ft		fs	M	1	61	**cabs.nge.d M,fs,ft**	3D
17	22	ft		fs	M	1	61	**cabs.nge.ps M,fs,ft**	3D
17	16	ft		fs	M	1	62	**cabs.le.s M,fs,ft**	3D
17	17	ft		fs	M	1	62	**cabs.le.d M,fs,ft**	3D
17	22	ft		fs	M	1	62	**cabs.le.ps M,fs,ft**	3D
17	16	ft		fs	M	1	63	**cabs.ngt.s M,fs,ft**	3D
17	17	ft		fs	M	1	63	**cabs.ngt.d M,fs,ft**	3D
17	22	ft		fs	M	1	63	**cabs.ngt.ps M,fs,ft**	3D
18	0	t		cs	0		0	**mfc2 t,cs**	
18	1	t		cs	0		0	**dmfc2 t,cs**	MIPS64
18	2	t		cs	0		0	**cfc2 t,cs**	
18	3	t		cs	0		0	**mfhc2 t,cs**	R2
18	4	t		cs	0		0	**mtc2 t,cs**	
18	5	t		cs	0		0	**dmtc2 t,cs**	MIPS64

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
18	6	t		cs	0		0	**ctc2 t,cs**	
18	7	t		cs	0		0	**mthc2 t,cs**	R2
18	8	0		broffset				**bc2f p**	
18	8	1		broffset				**bc2t p**	
18	8	2		broffset				**bc2fl p**	
18	8	3		broffset				**bc2tl p**	
19	b	t		0	fd		0	**lwxc1 fd,t(b)**	
19	b	t		0	fd		1	**ldxc1 fd,t(b)**	
19	b	t		fs	0		8	**swxc1 fd,t(b)**	
19	b	t		fs	0		9	**sdxc1 fd,t(b)**	
19	b	t		hint	0		15	**prefx hint,t(b)**	MIPS64 or R2
19	s	ft		fs	fd		30	**alnv.ps fd,fs,ft,s**	MIPS64 or R2
19	fr	ft		fs	fd		32	**madd.s fd,fr,fs,ft**	
19	fr	ft		fs	fd		33	**madd.d fd,fr,fs,ft**	
19	fr	ft		fs	fd		38	**madd.ps fd,fr,fs,ft**	PS
19	fr	ft		fs	fd		40	**msub.s fd,fr,fs,ft**	
19	fr	ft		fs	fd		41	**msub.d fd,fr,fs,ft**	
19	fr	ft		fs	fd		46	**msub.ps fd,fr,fs,ft**	PS
19	fr	ft		fs	fd		48	**nmadd.s fd,fr,fs,ft**	
19	fr	ft		fs	fd		49	**nmadd.d fd,fr,fs,ft**	
19	fr	ft		fs	fd		54	**nmadd.ps fd,fr,fs,ft**	PS
19	fr	ft		fs	fd		56	**nmsub.s fd,fr,fs,ft**	
19	fr	ft		fs	fd		57	**nmsub.d fd,fr,fs,ft**	
19	fr	ft		fs	fd		62	**nmsub.ps fd,fr,fs,ft**	PS
20	s	t		broffset				**beql s,t,p**	
21	s	t		broffset				**bnel s,t,p**	
22	s	0		broffset				**blezl s,p**	
23	s	0		broffset				**bgtzl s,p**	
24	s	d		(signed) const				**daddi d,s,const**	MIPS64
25	s	d		(signed) const				**daddiu d,s,const**	MIPS64
26	b	t		offset				**ldl t,o(b)**	MIPS64

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
27	b	t		offset				**ldr t,o(b)**	MIPS64
28	s	t		0	0		0	**madd s,t**	
28	s	t		d	0		0	**madd d,s,t**	R3900
28	s	t		0	0		1	**maddu s,t**	
28	s	t		d	0		2	**mul d,s,t**	
28	s	t		0	0		4	**msub s,t**	
28	s	t		0	0		5	**msubu s,t**	
28	s	s		d	0		32	**clz d,s**	
28	s	s		d	0		33	**clo d,s**	
28	s	s		d	0		36	**dclz d,s**	
28	s	s		d	0		37	**dclo d,s**	
28	code						32	**sdbbp code**	EJTAG
29	target							**jalx target**	MIPS16e
31	s	t		sz	pos		0	**ext t,s,pos,sz**	R2
31	s	t		sz	pos		1	**dextm t,s,pos,sz**	MIP64R2
31	s	t		sz	pos		2	**dextu t,s,pos,sz**	MIP64R2
31	s	t		sz	pos		3	**dext t,s,pos,sz**	MIP64R2
31	s	t		sz	pos		4	**ins t,s,pos,sz**	R2
31	s	t		sz	pos		5	**dinsm t,s,pos,sz**	MIP64R2
31	s	t		sz	pos		6	**dinsu t,s,pos,sz**	MIP64R2
31	s	t		sz	pos		7	**dins t,s,pos,sz**	MIP64R2
31	0	t		d	2		32	**wsbh d,t**	R2
31	0	t		d	16		32	**seb d,t**	R2
31	0	t		d	24		32	**seh d,t**	R2
31	0	t		d	2		36	**dsbh d,t**	MIPS64R2
31	0	t		d	5		36	**dshd d,t**	MIPS64R2
31	0	t		hwr	0		59	**rdhwr t,hwr**	R2
32	b	t		offset				**lb t,o(b)**	

表 8.6: 续

31-26	25-21	20-18	17-16	15-11	10-8	7-6	5-0	汇编名	所属 ISA
33	b	t		offset				**lh t,o(b)**	
34	b	t		offset				**lwl t,o(b)**	
35	b	t		offset				**lw t,o(b)**	
36	b	t		offset				**lbu t,o(b)**	
37	b	t		offset				**lhu t,o(b)**	
38	b	t		offset				**lwr t,o(b)**	
39	b	t		offset				**lwu t,o(b)**	MIPS64
40	b	t		offset				**sb t,o(b)**	
41	b	t		offset				**sh t,o(b)**	
42	b	t		offset				**swl t,o(b)**	
43	b	t		offset				**sw t,o(b)**	
44	b	t		offset				**sdl t,o(b)**	MIPS64
45	b	t		offset				**sdr t,o(b)**	MIPS64
46	b	t		offset				**swr t,o(b)**	
47	b	op		offset				**cache op,o(b)**	
48	b	t		offset				**ll t,o(b)**	
49	b	ft		offset				**l.s t,o(b)**	
50	b	cd		offset				**lwc2 cd,o(b)**	
51	b	hint		offset				**pref hint,o(b)**	
52	b	t		offset				**lld t,o(b)**	MIPS64
53	b	ft		offset				**l.d ft,o(b)**	
54	b	cd		offset				**ldc2 cd,o(b)**	
55	b	t		offset				**ld t,o(b)**	MIPS64
56	b	t		offset				**sc t,o(b)**	
57	b	ft		offset				**s.s ft,o(b)**	
57	b	ft		offset				**swc1 ft,o(b)**	
58	b	cs		offset				**swc2 cs,o(b)**	
60	b	t		offset				**scd t,o(b)**	MIPS64
61	b	ft		offset				**s.d ft,o(b)**	
61	b	ft		offset				**sdc1 ft,o(b)**	
62	b	cs		offset				**sdc2 cs,o(b)**	
63	b	t		offset				**sd t,o(b)**	MIPS64

8.6.2 指令编码表的几点注释

- 指令别名：大多数情况下，我们对同一指令只保留一个助记符，但是偶尔会有例外。像 **nop** 和 **l.s** 这样的指令是如此的普遍，以至于包括进来比省略反而更简单。
- 协处理器指令：曾经定义过的但是现在不用的指令被清除了。协处理器 3 在 MIPS I CPU 中从来没用过，而且和 MIPS32/64 的浮点单元不兼容——其中有些已被回收用于不同用途而会成为标准协处理器操作码，其中包括存储器加载在内。

8.6.3 编码和简单实现

如果你看一下指令的编码，有时可以看出 CPU 是怎样设计的。尽管有各种不同的编码，在流水线很早期就需要的域以一种非常有规律的方式编码。

- 源寄存器总是位于同样的位置，所以 CPU 可以从整数寄存器池取出两个操作数而无需条件译码。在某些指令中，两个寄存器都不需要，但是既然寄存器池设计来就是为了在每个时钟周期上提供两个源操作数的，也没有什么损失。
- 16 位常数总是位于同样的位置，允许适当的指令位被直接送进 ALU 的输入选择器而不用条件移位。

8.7 指令按功能分类

我们按以下顺序把指令集分成了合理大小的块：

- 空操作 No-op
- 寄存器/寄存器传输：用得很广，如果算不上亮点的话；包括条件传送在内
- 常数加载：作为数值和地址的整型立即数
- 算术/逻辑指令
- 整数乘法、除法和求余数
- 整数乘加
- 加载和存储
- 跳转、子程序调用和分支
- 断点和自陷

- CP0 功能：CPU 控制指令
- 浮点
- 用户态下对“地下”特性的受限访问：**rdhwr** 和 **synci**

8.7.1 空操作

nop: MIPS 指令集空操作 nop 很多，因为任何以 **zero** 为目标的指令都不做任何事情。最常用的一个是 **sll zero, zero, 0**，其二进制编码恰好为全零的字。

ssnop: 另外一个空操作，其编码就是 **sll zero, zero, 1**。

这个指令不得与其它指令同时发送，这样就保证了其运行要花费至少一个周期时间。这在简单的流水线的 CPU 上无关紧要，但在复杂些的实现上对于实现强制的编程延时很有用。

8.7.2 寄存器/寄存器传送

move: 通常用跟 **$zero** 寄存器的 or 来实现。一些个别 CPU ——由于某种原因对加法的支持比对逻辑运算更好——用 **addu**。

条件传送

对于减少分支有用的替代指令（参见第 8.5.3）。

movf, movt: 通过测试浮点条件码在整数寄存器进行条件传送。

movn, movz: 根据另一个寄存器的状态对整数寄存器进行条件传送。

8.7.3 常数加载

dla、**la**: 用来加载程序中某些带标号的位置或变量的地址的宏指令。当用 64 位指针时（只有在大型的 Unix 之类的系统中才会用到）你只要 **dla** 就够了。这些指令接受和所有其它 load 和 store 同样的寻址模式（尽管确实要求不同的东西）。

dli、**li**: 装入立即数常数。**dli** 是 64 位版本，并不是所有工具链都支持，只在加载太大以至于 32 位装不下的无符号数的时候才需要。这是一个宏指令，根据具体的常数大小展开成为几条指令。

lui: 把立即数加载到寄存器高位。16 位常数加载到寄存器的第 16–31 位，位 32–63（如果有的话）设置为与第 31 位相同，低位 0–15 清零。该指令是加载任意 32 位常数的一对机器指令中的一个。汇编程序员可能永远不会直接写这个指令；而由 **li** (load immediate) 和 **la** (load address) 之类的宏指令使用，最主要的是用于实现各种有用的寻址模式。

8.7.4 算术/逻辑运算

算术/逻辑运算指令进一步可以分解为下列类型：

加法

add、**addi**、**dadd**、**daddi**: **addu** 指令的很生僻的形式，当结果溢出时会自陷。可能对于 COBOL 编译器有用。

addu、**addiu**、**daddu**、**daddiu**: 分别用于 32 位和 64 位的加法。在这里以及整个指令集中，64 位版本的指令开头用“d”（表示 doubleword）表示；还有你无需指定“立即”助记符——只要给汇编器一个常数就行了。如果你要的常数用指令的 16 位域表示不下，那么汇编器将生成一列指令。

dsub、**sub**: 溢出时自陷的减法变体。

dsubu、**subu**: 常规的 64- 和 32- 位减法（没有减去立即数的指令，当然是因为加法中的立即数可以为负值）。

其它算术指令

abs、**dabs**: 绝对值；展开为设置和分支（或条件传送如果有的话）指令。

dneg、**neg**、**dnegu**、**negu**: 单目，取相反数；不带 U 的助记符溢出时会自陷。

逐位逻辑操作指令

and、**andi**、**or**、**ori**、**xor**、**xori**、**nor**: 三操作数逐位逻辑操作。不用写“立即数” 类型——当给一个常数操作数的时候，汇编器会自动生成。注意没有 **nori** 指令。

not: 用 **nor** 实现的双操作数指令。

移位和循环移位

drol、**dror**、**rol**、**ror**: 循环右移或左移；展开成四条指令的序列。

dsll、**dsll32**、**dsllv**: 64 位（双字）左移，低位补零。三个不同指令提供三种指定移位量的不同方式：常数 0–31 位，常数 32–63 位，或者用另一个寄存器低 6 位的内容。汇编程序员只要写 **dsll** 助记符就行了。

dsra、**dsra32**、**dsrav**: 64 位（双字）算术右移。称为“算术”是指把位 63——符号位——复制到高位。那就是说当用于有符号的 64 位整数时正确的实现了除以 2 的方幂。编程时总是写 **dsra** 助记符；汇编器会根据指定的移位量选择指令格式。

dsrl、**dsrl32**、**dsrlv**: 64 位（双字）逻辑右移。这个称为“逻辑”是因为把 0 填充到高位。虽然有三个不同的指令，汇编程序员永远应当采用 **dsrl** 这个助记符；汇编器会根据指定的移位量选择指令格式。

sll、**sllv**: 32 位左移。你只要写 **sll** 助记符就行了。

sra、**srav**: 算术右移（用符号位补齐高位）。只要写 **sra** 就行了。

srl, srlv: 逻辑右移（高位用 0 补齐）。只要写 srl 就行了。

条件设置指令

slt、**slti**、**sltiu**、**sltu**: 硬件指令，如果条件满足就写个 1，否则写个 0。只要写 **slt** 或者 **sltu**。

seq、**sge**、**sgeu**、**sgt**、**sgtu**、**sle**、**sleu**、**sne**: 根据更加复杂的条件设置目标操作数的宏指令。

8.7.5 整数乘法、除法和求余数

整数乘除的机器指令不同于普通指令，因为 MIPS 的乘法器是一个独立的单元，没有集成进正常的流水线中去，执行起来比正常的整数指令要花费更多的时间。有一些机器指令来启动乘法或者除法，然后乘除运算就和后继指令并行进行。

同一个乘法单元也提供整数乘累加和乘加之令（参见第 8.7.6 节）。

用单独的硬件进行处理的一个结果就是，乘除指令并不包括溢出和除零测试（因为异步运行不能产生异常）而且通常不把结果存进通用寄存器（与后续指令竞争写寄存器池会把流水线复杂化）。乘法/除法的结果出现在两个单独的寄存器 **hi** 和 **lo** 中。你只能用两条特殊指令 **mfhi** 和 **mflo** 来访问这些值。即使在最早期的 MIPS CPU 中，结果寄存器也是互锁的：如果你试图在结果出来之前读取，CPU 就会停下来等待数据到达。

但是，当你书写通常乘法/除法的汇编助记符的时候，汇编器会生成一个指令序列来模拟一个三操作数的指令并进行溢出检查。一条 **div**（有符号数除法）可能展开成为 13 条指令。额外的指令通常置于开始除法操作的 **div** 和提取结果的 **mflo** 之间。额外的指令看上去效率不高，但是它们和硬件除法器是并行运行的；在大多数 MIPS CPU 上除法运算本身要花费 7–75 个周期。

MIPS 公司的汇编器会把乘以一个常数或除以 2 的常数次幂的计算转换成为适当的移位、掩码等等。但是大多数工具链中的汇编器可能把这个任务留给编译器去做。

按照一个不太明显的约定，写的时候以寄存器 **zero** 作为结果的乘法或除法将生成原始的机器指令。[5] 此后从 **hi** 或 **lo** 中取出结果并进行必要的检查，就是你的事了。

下面是乘除指令的完整列表。

[5] 有些工具箱解释特殊的助记符，**mult** 表示乘法，**divd** 表示除法的机器指令。然而，指定 zero 作为结果寄存器的做法虽然奇怪，但更易于移植。

ddiv、**ddivu**、**div**、**divu**: 整数除法的三操作数宏指令，带有 64-/32-位和有符号/无符号的选项。在除以零的时候都会自陷；有符号指令溢出时会自陷。用 **zero** 作为目标寄存器就得到启动除法的机器指令。

ddivd、**ddivdu**、**divd**、**divdu**: 有某些工具链提供的原始机器指令助记符。最好用 **ddiv zero, ...**。

divo、**divou**: 带溢出检查的除法的名字，但实际上和写成 **div**、**divu** 是一样的。

dmul、**mul**: 三操作数 64-/32- 位乘法指令。没有溢出检查；因而不需要该宏指令的无符号版本——截断的结果对于有符号和无符号解释都是一样的。汇编器如果知道正在为 MIPS32 之前未实现这些指令的 CPU 在汇编的话，将会展开成等价的宏。

mulo、**mulou**、**dmulo**、**dmulou**: 乘法宏指令，如果结果超出一个通用寄存器能够容纳的范围就会自陷。

dmult、**dmultu**、**mult**、**multu**: 开始乘法的机器指令，有符号和无符号以及 32 位和 64 位的变体。结果从来不会溢出，因为分别有 64 位和 128 位的结果。结果的最低有效位存放在 **lo** 中，最高有效位部分存放在 **hi** 中。

drem、**dremu**、**rem**、**remu:** 余数操作，通过在除法后接一条 **mfhi** 实现。余数保存在 **hi** 寄存器中。

mfhi、**mflo**、**mthi**、**mtlo**: 从 **hi** 等寄存器传输数据。这些是访问整数乘除单元结果寄存器 **hi** 和 **lo** 的指令。如果你坚持使用自己提取结果的合成指令 **mul** 和 **div**，那么正常代码中你不需要写 **mflo/mfhi** 指令。

MIPS 的整数乘法，**mult** 或 **multu**，总是生成通用寄存器双倍宽度的结果，消除了溢出的可能性。结果中相当于寄存器宽度的高位和低位部分分别在 **hi** 和 **lo** 中返回。

除法操作将结果置入 **lo** 寄存器，整数余数置入 **hi**。**mthi** 和 **mtlo** 只有在异常之后恢复 CPU 状态的时候才用到。

8.7.6 整数乘（累）加

有些 MIPS CPU 有各种形式的整数乘累加指令——-没有一条属于 MIPS 的标准指令集。这些指令都接受两个通用寄存器作为源操作数，把结果累加到 **lo** 和 **hi** 中。与通常一样，“u”代表无符号的变体，但除此之外助记符（还有指令代码）都由具体的 CPU 实现决定。

dmadd16、**madd16**: NEC Vr4100 特有的指令，这些变体通过只接受 16 位操作数获得了速度，限制在 C 编译器中的使用。**dmadd16** 在 64 位的 **lo** 寄存器中累加 64 位结果。

mad、**madu**: Toshiba 的 R3900、IDT 的 R4640/4650 和 QED 的 CPU 中可见到这些指令，它们接受两个 32 位源操作数，把累加的 64 位结果存放在 **lo**

和 **hi** 寄存器中。Toshiba R3900 还允许三操作数的版本 **mad d,s,t**，其中累加的结果还传输到通用寄存器 **d** 中。

8.7.7 加载和存储

本小节列出所有汇编的整数加载/存储指令，以及其它寻址内存的操作。以下几点要引起注意：

- 对于所支持的不同的数据宽度有不同的指令：8 位（字节）、16 位（半字）、32 位（字）和 64 位（双字）。
- 对于小于机器寄存器宽度的数据类型，可以选择零扩展（前缀“u”代表无符号）或者符号扩展的操作。
- 这里列出的所有操作都可以写成汇编器支持的任意寻址模式（参见第 9.4 节）。
- 存储指令书写的时候先写源寄存器，然后写地址寄存器，和加载指令的语法保持一致；这点打破了 MIPS 指令中目标寄存器在先的一般规则。
- 机器加载指令要求数据自然对齐（半字对齐到两字节边界，字对齐到四字节边界，双字对齐到八字节边界）。但是汇编器支持一整套加载可能未对齐的数据的宏指令，这些指令都有一个“u”前缀（代表 unaligned）。

  所有声明为标准 C 程序的一部分的数据结构都会正确对齐。但是你可能碰到在运行时计算地址的未对齐数据、采用非标准的语言扩展声明的数据、从外部文件读入的数据等等。
- 每个加载指令交付结果都比计算指令在流水线中晚至少一个周期。对于任何 MIPS CPU，通过用一条有用的但没有依赖的指令填充延迟槽都会让效率最大化。

  在最老的（MIPS I）CPU 上要求程序员在每条加载指令之后保证一个时钟周期的延迟：认识 MIPS I CPU 的汇编器必要时会通过插入一条 **nop** 指令自动做到这一点。

下面是一个指令列表。

lb、**lbu**: 加载字节然后分别符号扩展或者零扩展到整个寄存器。

ld: 加载双字（64 位）。这条机器指令只在 64 位 CPU 上才有，但是针对 32 位目标的汇编器常常用一个从存储器加载 64 位到两个相邻的整数寄存器的宏指令来实现。这大概是一个十足的馊主意，但是有人就是想要某种兼容性。

ldl、**ldr**、**lwl**、**lwr**、**sdl**、**sdr**、**swl**、**swr**: 向左/向右、加载/存储、字/双字的版本。成对使用可以象 **ulw** 一样实现非对齐的加载/存储操作，当然你总可以自己来实现非对齐加载/存储（参见第 8.5.1 节）。

lh, lhu: 加载半字（16 位），然后分别符号扩展或零扩展到整个寄存器。

ll、**lld**、**sc**、**scd**: 连锁加载和条件存储（32 位和 64 位版本）；用于信号量的奇怪指令（参见第 8.5.2 节）。

lw、**lwu**: 加载字（32 位），然后分别符号扩展或零扩展到整个寄存器。**lwu** 只在 64 位 CPU 上才有。

pref、**prefx**: 预取数据到高速缓存（见第 8.5.8 节）。这条指令在 MIPS III 和更早的 CPU 上没有，可能为空操作。尽管 **pref** 用的是通常的寻址模式，但是 **prefx** 加上了在单个指令中实现的寄存器+寄存器寻址方式。

sb: 存储字节（8 位）。

sd: 存储双字（64 位）。在 32 位 CPU 上可能是一个宏（把两个相邻的整数寄存器存储到一个 64 位的内存地址）。

sh: 存储半字（16 位）。

sw: 存储字（32 位）。

uld、**ulh**、**ulhu**、**ulw**、**usd**、**ush**、**usw**: 未对齐的加载/存储宏指令。双字和字的版本采用特殊的向左向右加载的宏指令实现；半字操作用字节存取、移位和 or 来实现。注意正常的延迟槽规则并不适用于构成未对齐操作的向左/向右加载；流水线的设计让它们能从头跑到尾。第 2.5.2 节对非对齐的加载及其使用有更多的描述。

浮点加载和存储

l.d、**l.s**、**s.d**、**s.s**: 加载/存储双精度（64 位格式）和单精度（32 位格式）。操作数必须对齐，这里没有给出非对齐的版本。在 32 位 CPU 上，**l.d** 和 **s.d** 为加载两个 32 位内存块到相邻的 FP 寄存器或者存储相邻的两个 FP 寄存器到两个 32 位内存块（见第 7.5 节）的宏指令。这些指令也叫做 **ldc1**、**lwc1**、**sdc1**、**swc1**(load/store word/double 到协处理器 1)，但不要这样写。

ldsc1、**lwxc1**、**sdxc1**、**swxc1**: 采用基址寄存器+ 偏移量 的寻址模式的加载和存储。在指令 **ldxc1 fd, i(b)** 中，完整的地址必须位于和基址寄存器 **b** 指向的同一块程序存储区，否则可能会坏事。

如果你的工具链接受 **l.d fd, i(b)** 这样的语法，那就直接用吧。

8.7.8 跳转、子程序调用和分支

MIPS 体系结构对这些指令采用如下的 Motorala 命名法：

- PC-相对寻址指令称为“分支”，绝对地址指令称为“跳转”。操作的助记符分别以 **b** 和 **j** 开头。
- 子程序调用为“跳转并链接”或“分支并链接”，助记符以 **al** 结尾。

- 所有的分支指令，甚至分支并链接指令，都是有条件的，测试一个或者两个寄存器。无条件的版本可以很容易的合成——比如，**beq $0, $0, label**。

j: 这条指令把控制无条件转移到一个绝对地址。实际上，**j** 并不怎么管理 32 位地址：目标地址的高 4 位并不是指令给出的，而是用当前 PC 的高 4 位的值。大多数时候这没多大关系；28 位地址仍然给出最大 256 MB 的代码大小。

要到更远的地方去，要借助于 **jr**（寄存器跳转）指令，该指令也用于跳转目标需要计算的情形。写的时候可以只写一个 **j** 助记符和一个寄存器，但是流行的做法不这样写。

jal、**jalr**: 这些实现了直接和间接的子程序调用。在跳转到指定地址的同时，还要保存返回地址（该指令的地址加上 8）到寄存器 **ra**，即 **$31** 的别名。[6]为什么给程序计数器加上 8？记得跳转指令和分支指令一样，总是执行紧跟在后面的分支延迟槽指令，所以返回地址必须为分支延迟槽之后的指令。子程序返回通过寄存器跳转完成，最常用的就是 **jr ra**。

PC-相对的寻址的子程序调用可以使用 **bal**、**bgezal** 和 **bltzal** 指令。条件分支并链接指令即使在条件错误的时候也把返回地址存入 **ra**，这在用当前指令地址进行计算的时候有用。

b: 相对 PC 的无条件（但是相对短程的）分支。

bal: 相对 PC 的函数调用。

bc0f、**bc0f1**、**bc0t**、**bc0t1**、**bc2f**、**bc2fl**、**bc2t**、**bc2t1**: 根据协处理器 0 和协处理器 2 的条件位进行分支，这两个条件位在大多数现代的 CPU 上都不存在。在老式的 CPU 上，这些指令测试一个输入引脚信号。

bc1f、**bc1f1**、**bc1t**、**bc1t1**: 根据浮点条件位（后来的 CPU 有多个浮点条件位）分支。

beq、**beq1**、**beqz**、**beqz1**、**bge**、**bge1**、**bgeu**、**bgeul**、**bgez**、**bgez1**、**bgt**、**bgt1**、**bgtu**、**bgtu1**、**ngtz**、**bgtz1**、**ble**、**ble1**、**bleu**、**bleul**、**blez**、**blez1**、**blt**、**blt1**、**bltu**、**bltu1**、**bltz**、**tltzl**、**bne**、**bnel**、**bnez**、**bnez1**: 全部双操作数或者单操作数的比较分支指令的列表，大多数为宏指令。

bgezal、**bgezall**、**bltzal**、**bltzall**: 条件函数调用的原始机器指令，如果有可能需要这样做的话。

8.7.9 断点和自陷

break: 导致一个“断点”类型的异常。由调试器在汇编器合成代码引发的自陷中用。

sdbbp: 引发第 12.1 节讲述的 EJTAG 异常的断点指令。

[6]实际上 **jalr** 指令允许你指定一个不是 **$31** 的寄存器保存返回地址，但很少这样用。如果你不指定汇编器会自动用 **$31**。

syscall: 引发一个传统上用于系统调用的异常类型。

teq、**teqi**、**tge**、**tgei**、**tgeiu**、**tgeu**、**tlt**、**tlti**、**tltiu**、**tltu**、**tne**、**tnei**: 条件异常，根据各种单、双操作数的条件。这个是给编译器和解释器用的，想要实现运行时数组边界检查之类的操作。

8.7.10 CP0 功能：CPU 控制指令

CP0 的功能可以分成以下几种类型：

数据传送

cfc0、**ctc0**: 把数据移进移出 CP0 控制寄存器，到目前为止的 MIPS CPU 都还没有这样的控制寄存器。但是不久后的一天也许就有了这样的寄存器。

mfc0、**mtc0**、**dmfc0**、**dmtc0**: 在 CP0 寄存器和通用寄存器之间传送数据。

cfc2、**ctc2**、**dmfc2**、**dmtc2**、**mfc2**、**mtc2**: 协处理器 2（如果实现了的话）的指令。很少见。

用于 CPU 控制的特殊指令

eret: 从异常返回（见第 5 章）。

dret: 从异常返回（R6000 版本）。这条指令已过时，本书不再赘述。

rfe: 来自 MIPS I 的异常结束指令——实际上已经过时。令人好奇的是，**rfe** 仅仅恢复状态寄存器，通过被置于把控制转回到重新开始地址的一条 **jr** 指令的分支延迟槽内来得到执行。

cache: 第 4.9 节讲的多种形式的高速缓存控制指令。

sync: 对于可能乱序执行 load/store 的 CPU 的内存访问进行同步的指令（见第 8.5.9 节）。与本节讲的其它指令不同，它没有使用 CP0 指令编码，在用户态程序中可以合法使用。

tlbp、**tlbr**、**tlbwi**、**tlbwr**: 控制 TLB 即存储器地址转换硬件的指令（见第 6.3 节）。

standby、**suspend**: 进入省电模式（NEC Vr4100 CPU）。

8.7.11 浮点指令

在第 8.3 节列出了浮点指令。

8.7.12 用户态下对“地下”特性的有限访问

rdhwr(read hardware register): 允许用户特权级程序读取一些通常只有内核才可见的硬件信息。见第 8.5.12 节。

synci: 用户特权级指令。象 load/store 指令一样指定一个地址，作用于一个包含被寻址字节的高速缓存行大小的内存块上。当写入随后要执行的指令时用它。**synci** 执行必要的操作（常常意味着从 L1 一级数据高速缓存 D-cache 的回写，以及作废任何先前已经进入指令高速缓存 I-cache 的内容），保证 CPU 能够正确执行程序先前写入这块内存的指令。

sync: 存取防护指令，列在这里旨在强调它不是一条核心特权级的指令。该指令在第 8.5.9 节讲过。

第 9 章　阅读 MIPS 汇编语言代码

这一章将告诉你如何阅读 MIPS 的汇编代码——具体地说，就是为 GNU 汇编器 as 的 MIPS 版本所写的汇编代码，因为这是目前用得最广泛的 MIPS 汇编器。

因为在这里仅仅是一个介绍，我们将只考虑自 MIPS I 或 MIPS II 以来多年一直作为 ISA 一部分的常见的 32 位指令。

本章并不教你怎样编写汇编程序代码——这个题目本身就可以再写一本书了。绝大多数的读者很可能全部用 C 编程，只是偶尔会碰到一些现存的汇编程序要能看懂，或者只要做小小的修改；如果你对 MIPS 汇编的使用远远不止这个程度，你需要阅读伴随你的 MIPS 工具链的文档。

学习阅读 MIPS 汇编仅仅熟悉机器指令列表是不够的。这主要是因为以下几个原因:

- MIPS 汇编器提供了大量的预定义的宏指令，所以编译器的指令集要比实际的机器指令集大得多。上一章的列表包括了 GNU 工具链的汇编器认识的全部宏指令，[1]还有 CPU 硬件能够直接认识的机器指令。

- MIPS 汇编代码可以识别和解释一组用来管理汇编代码行为的、称为汇编“指示”或“伪操作”的特殊的关键字。例如，用特殊的关键字来标志程序函数代码的开始和结束、控制指令排列顺序、以及控制对代码的优化等等。

- 实际应用中，程序员人写的汇编代码往往要经过 C 语言预处理器的处理后，才被提交给汇编器，这几乎是通用的做法（尽管不是硬性强制的）。使用得当的话，可以写出具有极高可读性的汇编源代码，其中象机器寄存器之类的东西可以用对程序员有意义的名字来引用；把这些名字转换成汇编器要求的难以读懂的形式的任务最好留给工具链。

  许多程序员发现遵循这样一个简单约定会很方便：作为 C 语言预处理器输入的汇编源代码的文件名带有后缀“.S”，预处理后生成的输出的版本给一

[1]至少是由 MIPS 公司维护的汇编器截至 2005 年夏的版本所认识的宏指令，可能比当时公开的资料要早一些。

个同样的文件名但是后缀为“.s”。；这样使得管理两种不同形式的汇编源文件的任务更加容易。

在进一步阅读之前，先回头温习一下第二章的内容很有用，包括底层机器指令集、数据类型、寻址方式和传统的寄存器用法。即使你已经对那些材料很熟悉了，在那里额外插入一个书签也还有用。这样当我们继续往前看的时候，你可以很容易翻回去。

9.1 一个简单的例子

我们还是沿用在第 8 章见到的那个例子：C 语言库函数 `strcmp(1)` 的一个实现。不过这一次我们要包括汇编语法的基本成分，并且给出一些手工优化和调度的代码。这里给出的算法比原始的 `strcmp()` 函数要聪明一些；我们从下面的代码开始——仍然用 C ——用一种所有运算符都分开写以便于操纵的形式，代码如下：

```
strcmp(char* a0, char* a1)
{
    char t0, t1;
    while(1) {
        t0 = a0[0];
        a0 += 1;
        t1 = a1[0];
        a1 += 1;
        if (t0 == 0)
            break;
        if (t0 != t1)
            break;
    }
    return (t0 - t1);
}
```

这段初始形式的代码的执行时间会因为出现在每次循环中的两个条件分支（对应于两个 `if()` 语句）和两个加载（对应于两次数组下标操作）而受影响；每次分支和加载都引入一个延迟槽，而循环体中没有足够的工作来填满它们。此外，当代码沿着一对字符串进行时，在每对字节的比较（对应于 `while()` 语句结束的大括号）时都有一个强制的循环结尾分支。

即使在翻译成汇编语言之前，我们也可以对这个代码作有用的改进。最大的改变是展开循环让每次循环可以作两次比较；我们也可以把其中一次加载移到循环的结尾。通过这些变化，每次加载和分支的延迟槽都可以填上有用的工作：

```
int strcmp (char *a0, char *a1) {
    char t0, t1, t2;

    /* first load moved to loop end,
       so load for first iteration here */
    t0 = a0[0];

    while (1) {
        /* first byte */
        t1 = a1[0];
        if (t0 == 0)
            break;
        a0 += 2;
        if (t0 != t1)
            break;

        /* second byte */
        t2 = a0[-1];    /* we already incremented a0 */
        t1 = a1[1];     /* didn't increment a1 yet   */

        if (t2 == 0)
            /* label t21 in assembler */
            return t2 - t1;
        a1 += 2;
        if (t1 != t2)
            /* label t21 in assembler */
            return t0 = a0[0];
        t0 = a0[0];
    }
    /* label t01 in assembler */
    return t0 - t1;
}
```

我们已经从代码中消去了主要的低效代码，现在让我们把它翻译成 MIPS 汇编：

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF(strcmp)
```

```
    .set    noreorder
    lbu     t0, 0(a0)
1:  lbu     t1, 0(a1)
    beq     t0, zero, .t01          # load delay slot
    addu    a0, a0, 2               # branch delay slot
    bne     t0, t1, .t01
    lbu     t2, -1(a0)              # branch delay slot
    lbu     t1, 1(a1)               # load delay slot
    beq     t2, zero, .t21
    addu    a1, a1, 2               # branch delay slot
    beq     t2, t1, 1b
    lbu     t0, 0(a0)               # branch delay slot

.t21: j     ra
    subu    v0, t2, t1              # branch delay slot

.t01: j     ra
    subu    v0, t0, t1              # branch delay slot
    .set    reorder
END(strcmp)
```

上例中的注释帮助说明指令调度的方法；但是在进一步细看之前，我们应当解释一下上例中出现的许多新的语法结构。让我们按照出现的顺序逐个解释：

- `#include`: 该文件利用了 C 语言预处理器 cpp 来给常量起个有意义的名字，并定义一些简单的文本替换宏。这里，在把代码提交给汇编器之前，用 cpp 把两个头文件内嵌入汇编代码。`mips/asm.h` 定义了 LEAF 和 END（见下文），`mips/regdef.h` 定义了 **t0** 和 **a1** 等寄存器的习惯名称，可参阅第 2.2.1 节。

- 宏：这里我们用了 `mips/asm.h` 中定义的两个宏：LEAF 和 END。LEAF 的定义如下：

  ```
  #define  LEAF(name) \
          .text; \
          .globl    name; \
          .ent      name; \
  name:
  ```

  LEAF 被用来定义一个简单的例程（不调用其它例程，因而处于整个调用树的“叶子”位置——参见第 11.2.9 节）。非叶子（nonleaf）函数必须多做很多事情来保存寄存器、返回地址等等。除非你涉及到极为特殊的编程，否则不太可能碰到需要用汇编语言写非叶子函数的情况——几乎可以肯

定用 C 语言写这种函数更有意义，或许再由 C 语言调用一个封装真正需要用汇编写的代码的叶子函数。注意下面几点：

- **.text** 告诉汇编器，除非另有说明，应当把此后产生的代码直接放进目标文件中名叫“.text”的区中；从 C 语言编译生成的目标文件用同一个名字表示容纳所有代码的区。
- **.globl** 声明“name”为全局变量，该变量名要包括在模块的符号表内，而且名字在整个程序范围内必须是唯一的。这个类似于 C 编译器对函数名（如果不带 `static` 修饰符）的处理。
- **.ent** 对生成的代码没有影响，只是告诉汇编器将这一点标志为“name”函数的起始点，在调试记录中用到该信息。
- **.name** 在汇编器的输出中为该点提供一个名为“name”的标号，同时对“name”函数调用从该地址开始。

END 又定义了两个汇编项。

```
#define  END(name) \
        .size   name, .-name;  \
        .end    name
```

- **.size** 表示在符号表中，“name”和所用指令的字节数一道列出。
- **.end** 指出函数结尾，用于调试。

- **.set** 汇编指示：
  用来告诉汇编器怎样进行汇编。缺省时，MIPS 汇编器试图通过移动附近的指令尽量填充分支和加载延迟槽（但请不要担心——汇编器绝对不会做不安全的调整；如果找不到安全的方案，就会保持延迟槽不动）。大多数时候，这种行为是有用的，因为等于说你在写汇编程序的时候不必考虑填充延迟槽。

  但是如果我们的确需要精确控制指令顺序时怎么办，就像在许多用得极为频繁的库函数一类的情形？这就是 **.set noreorder** 的目的：告诉汇编器在下次碰到相对应的 **.set reorder** 之前停止重新排序。

  在这一对汇编指示之间包围的代码区域，我们告诉汇编器把产生的操作码按照在源代码中指令书写的次序写进目标文件。

- 标号：“1:”是数字标号，大多数汇编器都接受它作为局部标号。在一个程序里你可以有任意多个标号都为叫做“1:”；用“1f(forward)”引用下一个标号“1:”；用“1b(backward)”来引用上一个“1:”。这点非常有用。

- 指令：你可能注意到一些指令的顺序会出乎预料，因为 **.set noreorder** 这一指示暴露了下面的分支延迟槽，留给我们（为了效率）来保证刚刚加载的数据不会立即被下一条指令用到。

  比如说，在展开的循环的后半部分对寄存器 **t2** 的使用。必须要用第二个寄存器，因为 **lbu t2 -1(a0)** 位于前一条分支指令的延迟槽内，因而不能改写 **t0** ——如果发生分支，分支目标处的代码要用到 **t0** 中的值。

9.2 语法概要

过了一遍前面的例子让你看到了大多数重要的汇编指令在实际中是怎样使用的，也应当对你习惯汇编语言源代码文件书写 MIPS 指令的方式有所帮助。现在我们稍微系统地总结一下这些东西。如果你以前用过类似 Unix 的系统上的汇编器，那么主要的概念应该都不陌生。

9.2.1 布局、定界符和标识符

要了解这点，你得首先熟悉 C 语言。但阅读汇编代码的时候要注意下面几点：

- 汇编代码以行为单位，换行表示一个指令或指示的结束。你也可以在一行里写多条指令或指示，但是它们中间要用分号“;”隔开。
- 从“#”到行尾之间的内容为注释，汇编器将忽略它。但不要把“#”放在行的最左列：C 预处理器 *cpp* 对这样的行要特殊处理，容易导致混淆，而你可能会用到*cpp*。如果确定你的代码会经过 C 预处理器的预处理，那么可以在你的汇编代码中使用 C 风格的注释方式：`/*…*/`。只要你乐意，这种注释可以跨越多行。
- 标号和变量的名字可以是 C 语言里任意合法的标识符，同时还可以可以包含字符“`$`”和“`.`”。
- 在代码中你可以使用一个数（0 到 99 之间的十进制数）作为标号。常规的文本标号在文件中必须唯一，但是同一个数字标号可以在代码中重复使用任意多次。在分支指令中“1f”指向下一个“1:”，而“1b”指向前一个“1:”标号。这样就不用费心为那些很短的跳转和循环起名字了。把用名字命名的标号保留给子程序的入口点或者特别远的跳转。
- 强烈建议你使用表 2.1 中列出的 MIPS 寄存器的习惯命名；为此，你的源代码必须经过 C 预处理器进行预处理，源代码中要用 `#include` 包含一个头文件，名字可能是 `mips/regdef.h`。如果你决定不用预处理器，记住汇编器要求寄存器的名字写成美元符号加上数字的形式，比如 **$3** 代表通用寄存器 3，通用寄存器编号从 0 到 31。

- 没有和 C 语言中的“指针”直接对应的操作。当汇编器需要一个指针大小的值时，用地址来取代一个标号（或者其它可重定位的符号）。标识符“.”代表当前指令或者数据声明的地址。你甚至可以对这些东西做些有限的运算操作。
- 字符和字符串常数的定义方式与 C 相同。

9.3 指令的一般规则

MIPS 汇编器允许一些指令采用简便写法。提供的操作数可以少于机器码的要求，这时汇编器会解释为一个双操作数形式。或者在机器指令要求使用寄存器的地方用一个常数代替，这时汇编器能够推断出你需要的是该指令的立即数寻址的变体。本节总结一下常见的情况。

9.3.1 计算指令：三、二、一个寄存器

MIPS 执行计算的机器指令是三寄存器操作，就是说为带两个输入和一个输出的算术或逻辑函数，例如：

```
d = s + t
```

写成 **addu d, s, t**。

我们也提到过这里的三个寄存器可以重复。要生成一条 CISC 风格的双操作数指令，用一个和源操作数相同的目的寄存器就行了。如果你省略了 **s** 汇编器会自动帮你做：把 **addu d,s** 当作 **addu d,d,s** 同等看待。

neg 和 **not** 等单目运算符总是用一个或多个三个寄存器指令来合成的。汇编器期望这些指令最多有两个操作数，所以 **negu d,s** 等同于 **subu d, zero, s**，而 **not d** 被汇编成 **nor d,zero,d**。

可能最常见的寄存器到寄存器操作就是 **move d,s**。汇编器把这个无处不在的指令汇编为 **or d,zero,s**。

9.3.2 带立即数的运算指令

在汇编语言或者机器语言里，嵌入在指令中的常数被称为*立即数*。MIPS 的很多算术和逻辑指令都有另外一种用 16 位的立即数取代 **t** 寄存器的形式。立即数首先通过符号扩展或者零扩展扩展为 32 位，用哪种扩展取决于具体的指令。一般而言，算术运算指令进行符号扩展，而逻辑运算指令进行零扩展。

尽管立即数操作数生成不同于三寄存器操作数版本的机器指令（比如生成 **addui** 而不是 **addu**），但是程序员不需要明确区分。由汇编器检查最后一个操作数是一个寄存器还是一个立即数，并相应选择正确的指令：

```
addu    $2, $4, 64    ⇒    addiu   $2, $4, 64
```

如果立即数的值太大无法放进机器指令的 16 位操作数域，那么汇编器将再度出面帮忙。它自动将常数装进 汇编临时寄存器 **at/$1** 然后用它来进行操作：

```
addu    $4, 0x12345    ⇒    li      at, 0x12345
                            addu    $4, $4, at
```

注意 **li**（加载立即数）指令，在机器指令集里找不到的；**li** 是一个很常用的宏指令，它把一个任意的 32 位整数值加载进寄存器，程序员无需关心怎样放进去的——汇编器根据该整数值的性质自动选择最好的方式对操作编码。

如果 32 的值位于 ±32 KB 的范围内，汇编器会用一条 **addiu** 和 **$0** 相加；当高 16–32 位都为零的时候，可以用 **ori**；当低 0–15 位全为零的时候，就会用 **lui**；当这些都不是的时候，会选择一对 **lui/ori**：

```
li      $3, -5         ⇒    addiu   $3, $0, -5
li      $4, 0x8000     ⇒    ori     $4, $0, 0x8000
li      $5, 0x120000   ⇒    lui     $5, $0, 0x12
li      $6, 0x12345    ⇒    lui     $6, 0x1
                            ori     $6, $6, 0x2345
```

如果你在使用 **.set noreorder** 指示来控制分支延迟槽的管理，那么扩展成多条机器指令的汇编指令会有问题。如果在一个延迟槽中用了一个多条指令的宏，汇编器应该会警告。

9.3.3 关于 32/64 位指令

我们在前面（2.7.3 节）看到，MIPS 体系结构在扩展到 64 位时极为小心地保证即使是运行于 MIPS64 的机器上，MIPS32 程序的行为也不变；在 MIPS64 机器上，MIPS32 指令执行总是保持通用寄存器的高 32 位为全零或者全一（取决于位 31 的值）。

许多 32 位的指令可以直接用于 64 位系统上——比如全部的逐位逻辑运算——但是算术运算不行。加减乘除和移位都需要新的版本。新指令的命名是在老名字前面加上 **d** (double) 前缀：例如，32 位加法指令 **addu** 增强为新指令 **daddu** 来完成全 64 位精度算术运算。指令助记符前面的“d”通常是“double”的意思。

9.4 寻址模式

前面提到过，硬件只支持一种寻址模式：寄存器基地址+立即数偏移量（*base_reg+offset*），偏移量 *offset* 必须在 -32768 到 +32767 之间。但是汇编器能够合成代码来访问用各种别的方式指定的地址处的数据。这些方式包括：

- Direct（直接）：由你提供的数据标号或外部变量名。

- Direct+index（直接加索引）：一个偏移量，加上由寄存器指出的标号地址。
- Constant（常数）：一个解释为 32 位绝对地址的常数。
- Register indirect（寄存器间接）：是寄存器加偏移量，偏移量为零的特殊形式

上面的这些寻址方式，结合汇编器在编译时的一些简单的常数运算，加上宏处理器的使用，你就能够完成大多数要做的事情。下面是一些例子：

指令		展开为
`lw $2, ($3)`	⇒	`lw $2, 0($3)`
`lw $2, 8+4($3)`	⇒	`lw $2, 12($3)`
`lw $2, addr`	⇒	`lui at, %hi(addr)`
		`lw $2, %lo(addr)(at)`
`sw $2, addr($3)`	⇒	`lui at, %hi(addr)`
		`addu at, at, $3`
		`sw $2, %lo(addr)(at)`

上例中的 **addr** 符号可以是以下任何一种：

- 可重定位的符号——标号或者变量的名字（本模块内部或者别处）
- 可重定位的符号 ± 一个常数表达式（汇编器或者链接器可以在生成目标代码时处理这些）
- 32 位的常数表达式（例如设备寄存器的绝对地址）

这两个结构 **%hi()** 和 **%lo()** 代表地址的高 16 位和低 16 位。这不是简单的直接把地址分为高低两个半字，因为 **lw** 的 16 位偏移量域是解释为有符号数的。所以如果 **addr** 的值正好第 15 位为 1 ,那么 **%lo(addr)** 值就当作负值，我们需要增加 **%hi(addr)** 以补偿：

addr	**%hi(addr)**	**%lo(addr)**
0x1234 5678	0x1234	0x5678
0x1000 8000	0x1001	0x8000

la (load address) 宏指令对地址的服务，类似于 **li** 为整数常量提供的服务：

`la $2, 4($3)`	⇒	`addiu $2, $3, 4`
`la $2, addr`	⇒	`liu at, %hi(addr)`
		`addiu $2, at, %lo(addr)`

```
la      $2, addr($3)    ⇒       liu     at, %hi(addr)
                                addiu   $2, at, %lo(addr)
                                addu    $2, $2, $3
```

原则上，**la** 可以避免使用 **ori** 指令时在看上去像负值的 **%lo()** 值上导致的混乱。但是 load/store 指令有 16 位有符号的地址偏移量，结果链接器已经配备了对地址两部分进行修正的能力，以使相加后结果正确。所以 **la** 用了 **add** 指令以避免让链接要理解两种不同的修正类型。

9.4.1 相对于 GP 的寻址

MIPS 整个指令集都被塞在 32 位操作空间里的做法导致了一个直接后果就是，访问一个编译进来的存储器地址往往要花费至少两条指令，例如:

```
lw      $2, addr        ⇒       lui     at, %hi(addr)
                        ⇒       lw      $2, %lo(addr)(at)
```

在大量使用全局或静态数据的程序中，这往往导致最后编译出的代码臃肿而低效。

早期的 MIPS 编译器针对这个问题引入了一种修正技术，大多数的 MIPS 编译工具链都一直沿用了这一做法，通常被称为“全局指针 gp 相对寻址”。这个技术要求编译器、汇编器、链接器以及启动代码互相协同配合，把程序中的“小”变量和常数汇集到一块独立的内存区域；然后设置寄存器 **$28**（也称为*全局指针(global pointer)* 或 gp 寄存器）指向该区域的中央。（链接器生成一个特殊符号 **_gp**，其地址为该区间的中央。然后 **_gp** 的地址由程序的启动代码加载到 **gp** 寄存器，这一动作在任何 load/store 指令执行之前完成。）只要这些变量占用的空间加起来不超过 64 KB 大小，全部数据就都位于相对区域中点 32 KB 范围以内，这样一条 load 指令就变成：

```
lw      $2, addr        ⇒       lw      $2, addr - _gp(at)
```

问题就是编译器和汇编器要在单个模块编译的时候就必须确定哪些变量能够通过 **gp** 来访问。通常的做法把小于某个特定长度（通常缺省是 8 字节）的全部数据都放进该区间。这个上限长度通常可以通过编译器/汇编器的“`-G n`”选项来控制；指定“`-G 0`”将完全取消这种优化。

尽管这是一种非常有用的技巧，使用中有一些“陷阱”要注意。在写汇编代码时你必须特别注意对全局数据的声明要保持一致并且不要出错：

- 可写的要求初始化的小数据必须显式地置入 **.sdata** 区。
- 全局公共数据声明时必须指出其正确长度并且保持一致：

  ```
  .comm       smallobj, 4
  .comm       bigobj, 100
  ```

- 小的外部变量同样需要明确声明：

```
    extern      smallext, 4
```

- 对于大多数汇编器来说，除非变量的声明在变量的使用之前，否则就不理会声明。

在 C 语言中，全局变量必须在所有使用的模块中正确声明。对于外部数组，你可以象这样省略其长度：

```
extern int extarray[];
```

也可以给出其正确的长度：

```
extern int extarray[NARRAY];
```

有时候程序运行的方式决定了不能采用这种方法。有些实时操作系统（还有很多 PROM 监控程序）采用一块单独链接的代码来实现内核，应用程序使用远程的子程序调用来使用内核的功能。无法找到一个有效的方法在内核和应用程序分别使用的两个不同的 **gp** 值之间来回切换。这种情况下，要么应用程序、要么操作系统内核，两者之一（但没必要两个都这样做）必须使用“`-G 0`”选项来进行编译。

当使用“`-G 0`”选项编译任何一组模块的时候，通常那些需要与这些模块链接的所有库也都应该使用“`-G 0`”选项编译。如果链接器碰到几个模块对于给定的有名变量应当放在小数据区还是普通数据区意见不一的话，很可能会给出奇怪而毫无价值的错误信息。

9.5 目标文件及其在存储器映像中的布局

在本章的结尾简短地看一下程序在系统存储器中的布局方式，并提醒关于存储器布局和工具链产生的目标代码之间的关系的几点重要的注意事项。对于代码在装入内存之后看上去是个什么样子有个基本了解非常有用，特别是当你正在面对的任务是要让 MIPS 代码在一个新开发的系统硬件上第一次运行的时候。

MIPS 传统定义约定的代码和数据区（可置于 ROM 的程序）如图 9.1 所示。

在汇编程序中，各个区的选择按照下述的分组来组织。

.text、.rdata 和 .data

简单地把相应的区的名称放在数据和指令之前，如下例这样：

```
    .rdata
msg:.asciiz "Hello world!\\n"
    .data
table:
```

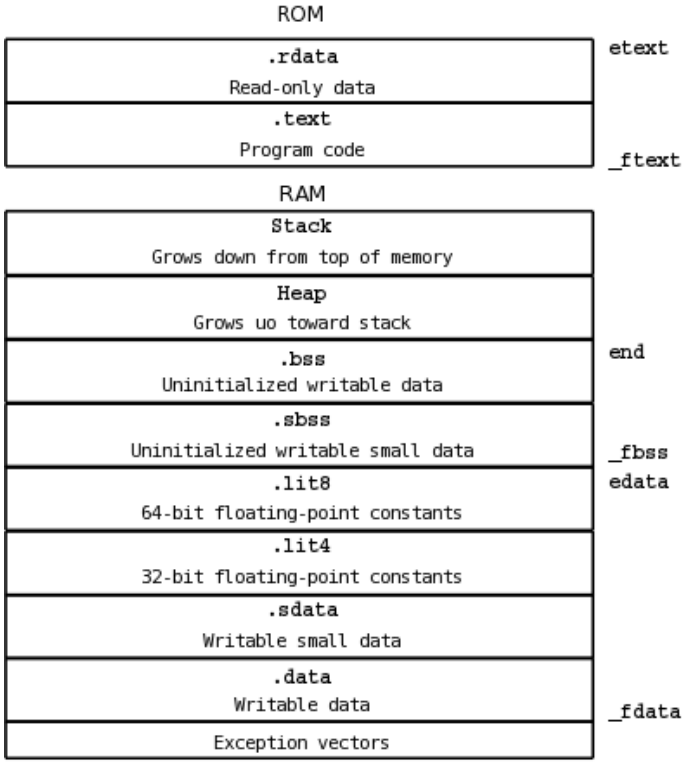

图 9.1: 可 ROM 化的目标代码段和典型的存储器布局

```
    .word 1
    .word 2
    .word 3

    .text
func:sub    sp, 64
    ...
```

.lit4 和 lit8：隐性浮点常数

你不能用汇编指示来写这些区的名称。它们是汇编器隐含生成的用来容纳作为 **li.s** 和 **li.d** 宏指令的参数的浮点常数。有些汇编器和链接器会合并相同的常数以节省空间。

如果应用程序构建是用 **gp** 相对寻址模式的话，**.lit4** 和 **.lit8** 可以放进“小数据”区。

.bss、.comm 和 .lcomm 数据

这些区的名字也不用作汇编指示。用来收集 C 模块中声明的所有静态或者

全局的未初始化的数据。C 语言有一个特性就是不同模块中的多个同名的定义是允许的，只要其中初始化的不超过一个就行。**.bss** 就是用来收集处处都没有初始化过的数据。FORTRAN 程序员会认出这就是称为 *common* 的公共数据块——这也是为什么这几个汇编指示这么起名的原因。

你应当总是指定数据的（以字节为单位的）空间大小。当程序链接的时候，这些数据将会得到足够容纳最大数据的空间。如果某个模块在初始化的数据区对该数据进行了声明，那么要采用其指定的所有大小和那个定义：

```
.comm   dbgflag, 4      # global common variable, 4 bytes
.lcomm  dbgflag, 4      # local common variable, 8 bytes
.lcomm  array, 100      # local common variable, 100 bytes
```

“非初始化”一词实际上用词不当。在 C 语言中，没有明确初始化的静态或者全局变量在程序开始的时候都会清零——这是操作系统或者程序启动代码的工作。

.sdata、小数据和 .sbss

想要把小的数据项分开的工具链把这些区作为上面的 **.data** 和 **.bss** 区的补充或替代方案。MIPS 处理器的工具链这样做是因为最后带来的小数据区足够紧凑，这样可以有一种如第 9.4.1 节所述的高效的访问机制，该机制依赖于在保留寄存器 **gp** 中维护一个数据指针。

注意 **.sbss** 不是一个合法的指示；如果数据项用 **.comm** 或 **.lcomm** 声明，并且占用空间小于传递给汇编程序的 `-G` 值，工具链就把该数据分配到 **.sbss** 区。

隐含的 **.lit4** 和 **.lit8** 常数区可能被放进小数据区，要看其上限的设置。

当采用 gp- 相对寻址时，**gp** 会被初始化为指向靠近“小数据区”中点的位置。

.section

开始一个任意命名的区，并提供控制标志（这与具体代码相关，还可能与具体工具包有关）。参考你的工具包手册，对于常见的区，永远用具体区名相应的汇编指示。

9.5.1 包括堆和栈在内的实际的程序布局

图 9.1 所示的程序布局适用于代码存放于 ROM 里，运行于一个裸 CPU 上（就是说，没有任何操作系统等中间软件提供的服务）的大多数实际系统。只读区可能被置于一块远离读写区的存储器中。

栈和堆作为系统地址空间的区域非常重要，但是汇编器或链接器对此的了解不同于对 .text 和 .data 等区。栈和堆是由运行时系统初始化和维护的。栈是

通过设置 **sp** 寄存器到可用内存的顶部（对齐到八字节的边界）来定义的。堆是用象 `malloc()` 等函数使用的一个全局变量指针来定义的；该变量常常初始化为 **end** 符号，是由链接器计算出的所有声明的变量使用的最高地址。

特殊符号

图 9.1 也给出了一些由链接器自动定义的特殊符号，帮助程序发现各个区的开头和结尾。它们来源于在 Unix 一类的操作系统中流传下来的习惯名称，其中有些是 MIPS 环境特有的。你的工具包可能定义了全部符号也可能没有；下表中那些打勾的几乎肯定都会定义：

符号	标准？	值
_ftext		文本（代码）区的开头
etext	✓	文本（代码）区的结尾
_fdata		初始化数据区的开头
edata	✓	初始化数据区的结尾
_fbss		未初始化数据区的开头
end	✓	未初始化数据区的结尾

第 10 章　向 MIPS 体系结构移植软件

很少有项目绝对要求所有的软件都要从头开发；大量的软件开发多多少少要用到某些已经现有的代码——在应用程序级，在操作系统级，都是如此。你可能会发现，你想要用在你的 MIPS 系统中的现有代码，最初是给其它的处理器家族开发的。当然了，你最少也需要重新编译一下源代码以生成用于 MIPS 上的二进制文件；但正如我们所见，这个任务可能远比重新编译来得复杂。可移植性指的就是一个软件被顺利无误地转换到一个新的环境，特别是一个新的指令集的难易程度。移植大量软件的工作绝非易事，如果待移植的软件是（或包括）操作系统，或是诸如设备驱动程序之类与操作系统密切相关的，移植的难度就会急剧上升。

高层软件（Linux 应用程序代码或类似代码）在写的时候至少都意识到了可移植性，并且很可能已经在几种不同环境中使用了，所以也许你无需任何修改只要重新编译一下就行了。底层软件——或许是某些嵌入式系统的源代码的大部分——问题就多了。完全在一种特定环境下开发出的软件很可能会出现移植性的问题，因为其开发者可能没有意识到要避免或者解决移植问题。本章的目标就是让你的注意力转移到在向 MIPS 移植软件的时候特别容易出问题的方面。

一个系统中驱动最底层硬件的部分不可避免的会有移植性问题；典型的嵌入式系统每一两年左右都要有重大的设计升级，在这种变化下仍然坚持要求保持原来的硬件/软件接口不变，就不合理了（当然也不会节省成本）。

10.1　MIPS 应用程序的底层软件：常见问题一览表

下面这些问题出现得相当频繁：

- 尾端：计算机世界分成了两大阵营，二者之间是一个不可逾越的鸿沟。大多数 MIPS CPU 可以设置成以大尾端或者小尾端运行；但即使你已经知道你的 MIPS 系统是怎样配置的，还是强烈建议你一定要透彻的理解了这个问题。在你之前已经有很多富有经验的程序员在这上面栽了跟头，以后还会有更多的人在这上面栽跟头。请参阅 10.2 节。

- 内存中数据的布局和对齐方式：你的程序对于 C 语言中声明的数据在内存中的布局有可能作了不适当（不可移植的）假设。使用 C `struct` 声明来映射文件数据或从通信链路上收到的数据几乎从来都是不可移植的。使用不同类型的指针或者联合体对私有数据的多个角度访问的程序中就潜伏有危险。

  然而，数据布局往往和其它（关于寄存器实用、参数传递、堆栈处理）约定相关，在下一章会讲到这些约定：如果你想要提前瞄一眼，参阅第 11.1 节。

- 直接管理高速缓存的需要：你也许会发现你想要复用的代码是在一种根本不实现高速缓存的处理器、或者是用了一个带有对于软件完全“透明”的高速缓存的 CPU（例如，在 PC 兼容的处理器上，几乎所有的高速缓存的副作用都被硬件隐藏）。但是大多数 MIPS CPU 为了保持硬件简单，让某些副作用保持可见并让软件负责高速缓存管理；我们将在第 10.3 节解释这样做意味着什么。

- 存储器访问顺序和重新排序：在许多现代的嵌入式系统或消费类系统中，系统周围传送的数据当从起点移动到最后的终点之间可能要链式通过多个子系统。这些子系统有可能自身封装了许多复杂的硬件，给你呈现出意料不到的问题。例如，在 CPU 和 I/O 设备之间传送的信息可能被迫进入队列等待，导致不同程度的延迟；或者也可能被分成几个独立的数据流，这样到达各自终点的顺序，不能保证和最初发送的顺序相匹配。第 10.4 节讨论这方面的一些典型问题及其解决方案。

- 用 C 语言书写程序：这与其说是问题不如说是机会。但是有些 MIPS 特有的东西，也能够用 C 写（也许应当优先于写汇编语言代码）。本节讲述内嵌汇编，存储器地址映射的寄存器，以及使用 MIPS 时可能遇到的杂七杂八的陷阱。

10.2 尾端：字、字节和位序

*尾端(endianness)*这个词是由 Danny Cohen(Cohen 1980) 引进计算机科学的。在一篇难得的极为幽默易读的文章中，Cohen 注意到，基于对通信系统中的字节地址和整数定义方式的人为选择，计算机体系结构已经分裂为两大阵营。

在 Johathan Swift 的《格列弗游记》中，“小端派”和“大端派”因为对于吃煮鸡蛋时应当先打哪一端才对的分歧而爆发了一场战争。Swift 在讽刺十八世纪的宗教争论，双方都看不到他们之间的区别完全是主观人为的。Cohen 的笑话被大家公认，此后就沿用下来了。尾端问题不仅仅与通信有关，还关系到可移植性。

计算机程序总是处理不同类型数据的序列和顺序：按顺序遍历一个字符串中的字符、一个数组的每个元素，或者二进制表示中的每个比特。C 程序

员普遍假定所有这些变量都存放在内存中，而内存本身就是一个字节的序列——`memcpy()` 可以拷贝任意数据类型。C 的 I/O 系统对所有的 I/O 操作都以字节为单位建模；你也可以 `read()` 或 `write()` 包含任意数据类型的存储块。

这样一个计算机可以写出一些数据，另一个计算机读取这些；突然间，我们对第二个计算机能否理解第一个计算机写的东西感兴趣了。

我们理解在处理对齐和填充的时候需要小心（细节详情参见第 11.1 节）。期望象浮点数那样的复杂数据类型总能够原封不动地传输是过于奢求了。但我们至少希望看到简单的以二进制补码形式表示的整数传输没问题；尾端的魔咒是整数也不行。十六进制的值 0x1234 5678 的 32 位整数读出时常常变成了 0x7856 3412 ——被进行了“字节交换”。要理解为什么会这样，我们先回头看一下。

10.2.1 比特、字节、字和整数

32 位的整数表示成一些二进制位的序列，每个位有不同的权重。最低有效位权重为 1，然后是 2 、4 ——就象十进制表示的“个、十、百、千”。当你的存储器以字节寻址时，32 位整数占用四个字节。对于各个字节怎样构成一个整数有两种合理的选择。有些计算机把最低有效位放在前面（即低地址的内存字节）有些把最高有效位放在前面——Cohen 称它们分别为小尾端和大尾端。当我 1976 年第一次接触计算机时，DEC 的小型机是小尾端的，而 IBM 的大型机是大尾端的；任何一方都不愿意让步。

值得强调的是这个选择的魔咒只在你能够寻址字节的时候才发作。1960 年代后期之前的先驱的计算机都是按照单个机器字为单位组织的：指令、整数、内存宽度都是同样的字大小。这样的计算机没有尾端问题：在存储器中以字为序，字内以位为序，二者没有关系。

就象打煮熟的鸡蛋一样，双方阵营都有充分的理由。

我们习惯于从左至右书写十进制数（看的时候通常也从左至右），读数时也是一样：莎士比亚可能会说“四和二十，”但我们则说“二十四。”这样写数的时候，自然的就先写最高位。在内存能以字节寻址之前，字节一开始出现只是作为一种把字符存放进字中的简便方式。1970 年代的那批 IBM 程序员花费了大半生的经历察看大量的输出列表，每组字符是代表一个表示数值的字。小尾端的数看上去很可笑，直观上是倾向大尾端的。但是写数的时候最高位在左边，字节地址沿同一方向增加，这样如果数的每个二进制位从右到左编号就会不一致：所以 IBM 把一个字的最高位标为 0 位。他们的世界如图 10.2 所示。

但是根据在整数类型内部的算术权重对各二进制位编号也很自然——即把位号 n 分配给算术权重为 2^n 的位置。这样在字节 0 存储位 0–7 就一致了，你又变成了小尾端派。在输出显示一个字时倒着写不太光彩，但是小尾端对于习惯于把内存当成一个字节序列的人来说特别有意义。尤其值得一提的是，Intel 是小尾端派。这样的字、字节和位看上去如图 10.1 所示。

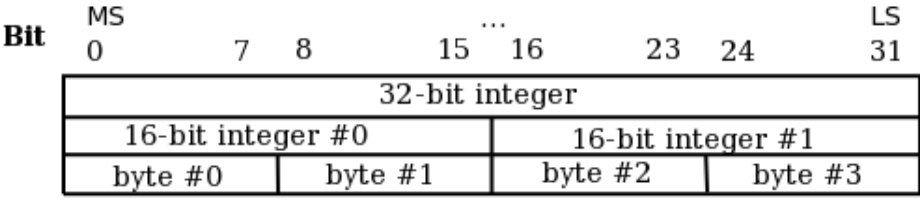

图 10.1: IBM（一致的大尾端）角度看到的位域

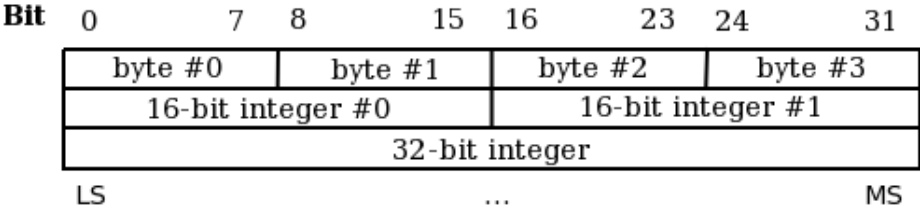

图 10.2: 一致的（Intel）小尾端角度看到的位域

你会注意到这些图中的内容完全相同：只是最高/最低位交换了，还有各个域的次序也反转了。IBM 大尾端派看到由字分解成字节，而小尾端派则看到由字节构成了字。这两种系统对于不同的人来说都是无可争辩的正确：二者都有许多优点，但是你只能从中选一。

让我们再回到刚才观察的问题。我们的被弄乱的字开始为 `0x1234 5678`，二进制就是 `00010010 00110100 01010110 01111000`。如果你直接就发送到具有相反尾端的系统上去，你肯定会看到所有的位都翻转了。遇到那种情况，你会接收到 `00011110 01101010 00101100 01001000`，十六进制为 `0x1E6A 2C48`。但是我们刚才说我们读到的十六进数为 `0x7856 3412`。

不错，位顺序的彻底倒转在某些情况下会出现；有一些通信链路先发送最高位，一些先发送最低位。但是在 1970 年代的某个时候，八位字节标准成了在计算机内部和计算机通信系统内部（称为“octet”）通用的基本单位。典型的通信系统从字节构建所有的报文，只有最底层的硬件工程师才知道哪个位先发送。

同时微处理器系统开始使用八位的外围控制器（更宽的控制器保留给高端产品），所有这些外围设备都有一个 8 位的端口从 0 到 7 编号，最高有效位为 7。不知道怎么搞的，没有发生明显的火并，每个字节都变成了小尾端，而且此后一直保持下来了。

早期的微处理器系统是在八位总线上用八位存储器系统的八位 CPU，所以没有尾端问题。Intel 的 8086 是 16 位小尾端的系统。当 Motorola 在 1978 年前后推出 68000 系列处理器时，他们非常推崇 IBM 的主干机体系结构。不论是出于对 IBM 的推崇，还是出于有意要跟 Intel 保持不同，他们认为自己也该用大尾端。但是 Motorola 无法抗拒已经占主导地位的字节内位次序的习惯约定——每

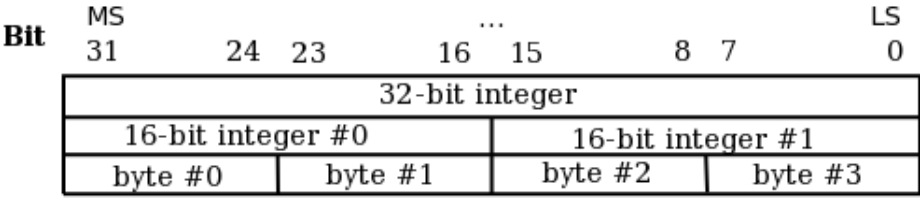

图 10.3: 从 68000 族处理器创始的不一致的大尾端角度看到的位域

个八位的 Motorola 外围设备连接到 68000 上的时候不得不反转数据总线。结果，68000 家族看上去如图 10.3 所示，位与字节以相反顺序编号。68000 及其后继产品继续在大多数成功的 UNIX 服务器和工作站（最明显的例子就是 Sun）上使用。当 MIPS 和其它 RISC 在 1980 年代刚出现的时候，其设计者为了吸引采用相应尾端的系统设计人员，他们设计的 CPU 可以用任一种尾端。自 68000 往后，大尾端就指 68000 式大尾端，位与字节方向相反。当你设置 MIPS CPU 为大尾端时，看上去就象图 10.3。这就是麻烦真正开始的地方。

当你阅读你的 CPU 硬件手册和看寄存器图时有一些小麻烦。每个人都认为寄存器（首先）是 32 位整数，所以都毫无二致的先画最高位（第 31 位）。这对程序员和硬件设计一类的人员有影响。该幅图导致了“左移”和“右移”指令的差别，还用以确定位域指令的位号参数，甚至影响构成 MIPS 指令的位域的标号。

一旦你碰到那种情况，当移植软件或者在不兼容的机器之间传送数据时就会有严重的软件问题；当连接不兼容的总线或者元件时就会有硬件问题。我们将分开讨论软件和硬件问题。

10.2.2 软件和尾端问题

这是一个面向软件的尾端定义：一个 CPU/编译器系统，当多字节整数最低地址的字节容纳的是最低有效位时就叫做小尾端；当多字节整数的最低地址的字节容纳的是最高有效位时就叫做大尾端。通过运行一段故意写得不可移植的代码，可以很容易地找出你的 CPU 是哪一种：

```
#include <stdio.h>

int main(void)
{
    union {
        int as_int;
        short as_short[2];
        char as_char[4];
    } either;
```

```
    either.as_int = 0x12345678;

    if (sizeof(int) == 4 && either.as_char[0] == 0x78) {
        printf("Little endian\n");
    }
    else if (sizeof(int) == 4 && either.as_char[0] = 0x12) {
        printf("Big endian\n");
    }
    else {
        printf("Confused\n");
    }
}
```

严格来讲，软件尾端是编译器工具链的一个属性，总可以——如果足够卖力工作的话——生成任何一种尾端的效果。但是在一个象 MIPS 这样可以寻址到字节的 CPU 上，内部就是用 32 位算术运算，如果强行跟硬件对抗会导致极端的低效，所以我们讨论 CPU 的尾端。

当然了，字节在地址空间布局的问题适用于除整数之外的其它数据类型；会影响到任何占用超过一个字节空间的数据，比如浮点数据类型、文本字符串甚至还有代表指令的 32 位操作码。算术权重的概念，对于其中一些非整数的数据类型只能有限制的使用，对于其它数据类型可能没有任何意义。

如果一种语言支持某些软件构造的数据类型，而这些类型超过硬件能管理的范围，那么这时尾端问题是纯粹的软件约定问题——可以用任何一种尾端构造。我希望现代的编译器作者能够理解这一点，最好让软件约定和硬件自己的约定保持一致。

尾端和程序可移植性

只要从来不从别处输入二进制数据，只要你避免以两种不同的整数类型存取同一块数据（就象上面的例子中有意做的那样），CPU 的尾端就是不可见的（你的代码就是可移植的）。现代的 C 编译器将尽量帮助你检测这种情况：如果你不小心这样做了，可能会得到编译器的错误或者警告。

但是你也许无法接受这些限制；你可能不得不处理从别处交付给你的系统的外来数据，或者要处理存储器映射的硬件寄存器。碰到这两个情况的任何一个，你都需要确切知道你的编译器是怎样访问内存的。

这一切看上去都没有什么危害，但是经验表明在所有数据映射问题中，尾端最容易引起混淆。我想这是因为即使是描述这个问题不带有偏向都很困难。这两种方案的最初起源就在于两种不同的画图和描述数据的方法；在不同的环境下二者都是很自然的。

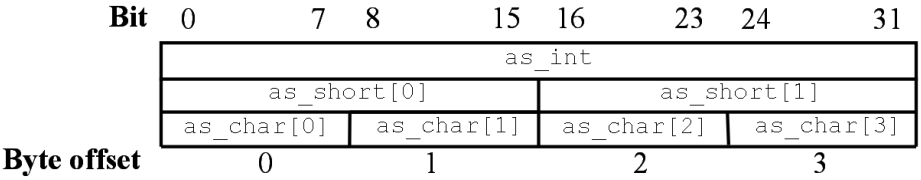

图 10.4: 典型的大尾端图像

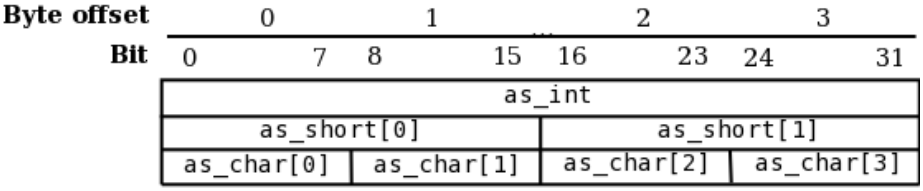

图 10.5: 小尾端图像

如上所见，典型的大尾端围绕着字来画图。所以给了我们一幅如图 10.4 里用的数据结构。要是按照 IBM 把最高位标记为 0 位的约定，图会好看得多，但是现在不再这样做了。

但是小尾端可能更强调从一种面向软件的、抽象的角度把计算机内存看作一个字节序列。所以同样的数据结构看上去像图 10.5。小尾端并不把计算机数据主要当成是数值，所以倾向于把低位的数（位、字节或者其它）写在左边。

不画图很难真正理解尾端问题，但许多人发现自己总是很难放下自己习惯的约定，例如，如果你习惯于从右到左对位编号，要花费很大的意志力才能从左到右编号（一个小尾端结构的图像让一个习惯于大尾端的人画出来看上去可能非常不合逻辑）。这是该题目容易引起混淆的本质：对一个不熟悉的约定即使是想象一下都很难不陷入自己熟悉的东西的影响。

10.2.3 硬件和尾端

我们在前面看到 CPU 内在的尾端，只有在同时提供对字长的数和更小粒度的以字节为单位的存储器系统的直接支持时才能体现出来。类似的，当字节寻址的系统与多字节宽度的总线连接时，硬件系统获得一个可以识别的尾端。

当你通过总线传输多字节的数据时，该数据的每个字节都有自己独立的地址。如果数据中最低地址的字节沿着最低位号的八根总线（字节通路）传播，该总线是 *小尾端的*。但是如果数据的最低地址的字节沿着最高位号的字节通路传输，那么该总线是 *大尾端的*。

在 CPU 的“内在”尾端及其作为总线的系统接口的尾端之间没有必然联系。但是，我不知道有哪个 CPU 的软件和接口的尾端不一样，所以我们可以说“CPU 的尾端”用来同时指其内部的组织和系统接口。

可以寻址到字节的 CPU 每次传输数据的时候都会表明自己是大尾端或者小尾端。Intel 和 DEC 的 CPU 是小尾端的；Motorola 680x0 和 IBM CPU 是大尾端的。MIPS CPU 在上电时可以配制成任意尾端；大多数其它的 RISC CPU 沿袭了 MIPS 的做法，让尾端成为可配置的——这在用一个新的 CPU 来升级现有系统的时候很方便。

硬件工程师通过对上位号来连接不同总线的做法不应该被指责。但是当你的系统包括尾端不相匹配的总线、CPU、或者外围设备时麻烦就来了。在这种情况下做出选择绝对不那么轻松；系统设计人员不得不“两害相权取其轻。”

- 保持位号一致/但是字节序被打乱：最明显的选择就是，设计人员可以按照位的编号把两个总线接起来，这样的效果是保持对齐的字里的位编号不变。但是因为位编号和字内的字节顺序相反，连线的两端看到的存储器的字节序列不同。

  和总线宽度不一致但是对齐到总线边界的数据在相连的总线上传输时会弄乱，每个总线宽度单元内部字节被交换。这种观感比软件问题更糟糕。软件中尾端错误的数据，找出数据类型的边界没什么问题；仅仅是数据本身没有意义。碰到这种硬件问题，数据边界也被搞乱了（除非碰巧，数据都对齐到总线宽度的“字”边界）。

  这里面有些蹊跷。如果经过接口传输的数据总是对齐到字边界的整数，那么位号一致的布线就会掩盖尾端的差别，避免了用软件转换整数的需要。但是硬件工程师很少确切知道在整个系统生命期内都会有什么样的数据经过接口传输。

- 字节地址一致/整数被打乱：设计人员可以选择通过把每个字节通路连接到相应的同样的字内字节地址从而保持字节地址不变，当然此时字节内的数据连线根本不匹配。那么至少整个系统对于把数据当成字节数组来看是一致的。

  然而，系统中可能还会有一些单元的软件尾端不匹配。所以一致的字节地址肯定会暴露出多字节整数数据表示的不一致现象。而且——特别是——即使是对齐到总线宽度边界的整数（传输的“自然”单元）当传送到另外的尾端时看上去也显得是字节交换了。

对于大多数目的，打乱字节地址更为有害，我们推荐“字节地址一致”的布线。当处理数据表示和传输问题时，程序员常常会回到 C 语言将内存当成是一个字节数组的模型上去，其它数据都是在此之上构建的。当你对内存的假定不对的时候，很难看出到底出了什么问题。

不幸的是，位号一致/字节地址打乱的连接从设计上看更为自然合理；要说服硬件工程师违反这个设计把事情做对可能非常困难。

并非系统的每一个连接都受到影响。假定我们有一个 32 位的存储器系统直接连上 CPU。CPU 的系统接口可能并没有包括字内字节地址——地址总线并不指定地址位 0 和 1。而是，许多 CPU 都有四个“字节选择”信号，表明正在传输的数据属于某个特定的字节通道。存储器阵列被连线到整个总线，写的时候，字节选择告诉存储器阵列该字内的四个可能字节中的哪个字节位置会被实际写入。在内部，CPU 把每个字节通道和字内字节地址相连，但是那跟存储器系统的操作无关。等效的说，存储器/CPU 构成的系统继承了 CPU 的尾端；只要 CPU 能够重新读回原来写入的数，字内字节 0 实际上在存储器的什么地方并不重要。[1]

很重要的一点就是，不要被 RAM 存储器这个有用的性质误导，就认为在简单的 CPU/RAM 系统中没有内在的尾端问题。你可以在宽总线上的任意传输中发现尾端问题。这里一个条件列表，在列出的条件下当构建一个存储器系统的时候，不能忽略 CPU 的尾端：

- 如果你的系统使用预先编程进 ROM 存储器的固件，在系统内硬件地址和字节通路连接的分配要和 ROM 编程时的方式相一致，ROM 中包含的数据需要匹配 CPU 配置成的尾端。等价于说，ROM 中的数据内容是从系统外部交付给系统使用的。如果要求代码直接从 ROM 中执行，正确的尾端就会特别重要，因为不可能让 CPU 在取指的时候对操作码进行软件字节交换。

- 当一个 DMA 设备到了要直接传输数据到存储器的时候，那么它对于尾端的概念就会起作用。

- 当 CPU 接口实际上不用字节选通，而是用一个字节宽度的代码（对 MIPS CPU 很常见）发送字内字节地址，那么至少译码 CPU 读写请求的硬件必须知道 CPU 用的是哪种尾端。如果 CPU 的尾端可以用软件配置，就会特别容易出错。

下一节让你告诉你的硬件工程师怎样设置一个字节地址一致的系统——甚至包括在用户可能以两种方式设置 MIPS CPU 时候，怎样让那个系统可以根据 CPU 配置。

尾端不一致的总线间接线

假定我们有一个配置成大尾端的 64 位的 MIPS CPU，我们需要把它连接到一个小尾端的 32 位总线上，比如 PCI 上。

图 10.6 显示了我们怎样连接数据总线以达到推荐的效果，就是在大尾端的 CPU 和小尾端的总线上看起来都一致的字节地址。

[1] 熟悉硬件的工程师可能会意识到这是一个更为一般的原则的结果：可写存储器阵列的一个性质就是，不管连接到它的地址和数据线怎么排列，都能继续工作。一个具体数据存在哪里并不重要，重要的是当你给出同样的读取地址时，你能够读到你原先写入的数据。

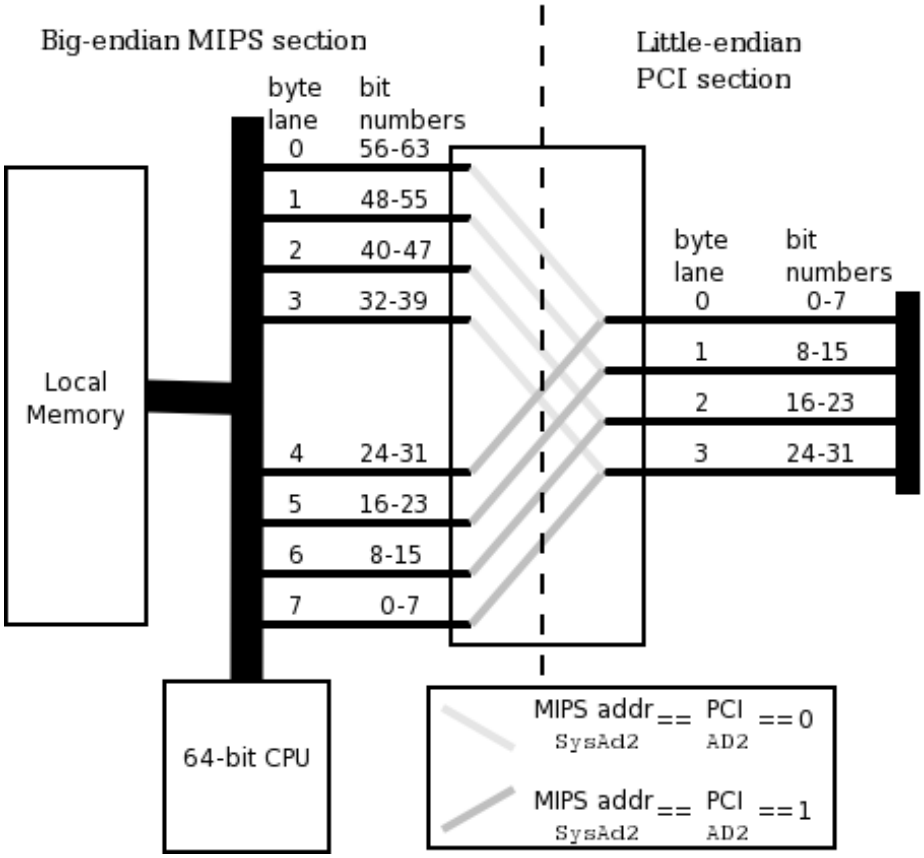

图 10.6: 大尾端 CPU 和小尾端总线之间的接线

图中的编号叫做“字节通道”显示了到达那里的字节数据的地址的总线宽度内部的字节部分。写进字节通道号是搞对一组连线的关键。

因为 CPU 总线是 64 位宽度而 PCI 总线是 32 位，你需要能够把宽总线的每一半根据“字”地址——即地址位 2 ——连接到窄总线，因为地址位 1 和 0 属于 32 位字内的字节地址。CPU 的 64 位总线是大尾端的，所以高位数据用的是低地址，这点可以从字节通道号上看出来。

你可能发现自己要瞪大眼睛在总线交换器周围的连接编号看好大一会儿，才能真正看出点门道来。这就是尾端带来的乐趣。

注意我只是给出了数据。PCI 数据和地址复用总线，在某些时钟周期，这些“字节通道” 传送的是地址。在地址周期，PCI 总线的 31 号线传送的是地址的最高有效位。在地址期间内与基于 MIPS 的系统的连接不应当被交换。

尾端可配置的接线

假定你想要做一块主板或者总线交换设备，允许 MIPS CPU 能以任一种尾端配置运行。上面的建议该做怎样的推广呢？

我们建议，如果你能够说服你的硬件工程师，你应当在 CPU 和 I/O 之间放置一个可编程的字节通道交换器。这种方法的工作原理如图 10.7 所示。注意这里只是一个 32 位的可配置的接口，把它推广到 64 位 CPU 连接的情形作为一个练习留给读者。

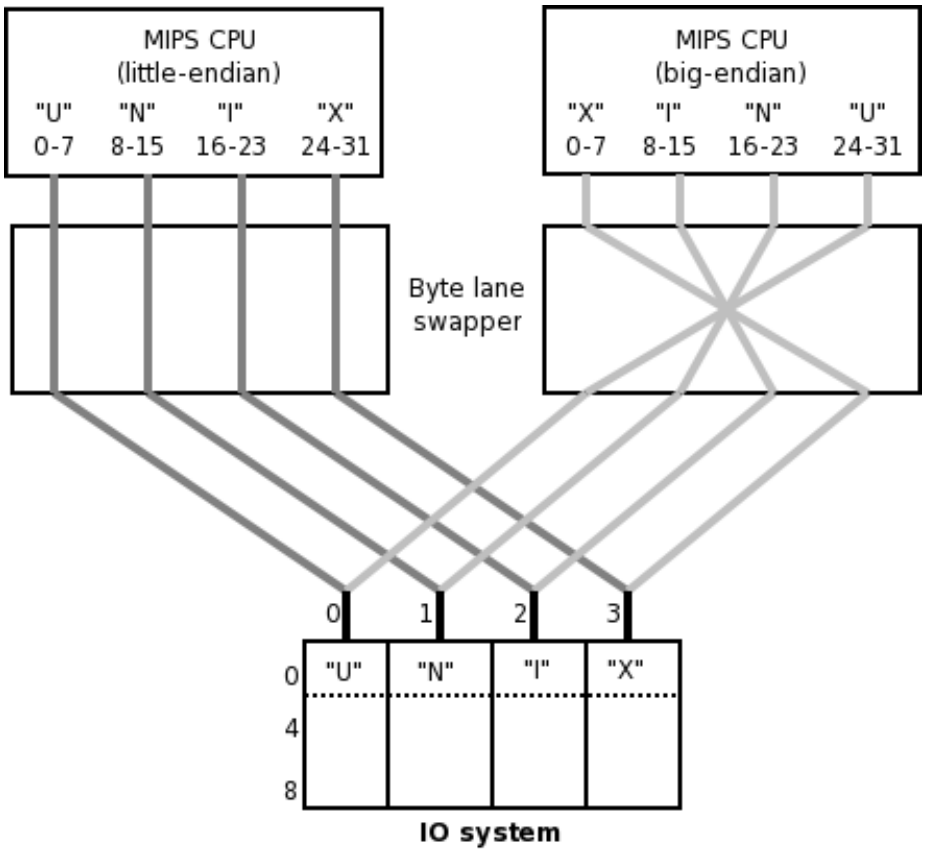

图 10.7: 字节通路交换器

我们把这个叫做字节通路交换器，而不是字节交换器，以强调其并不是根据每次传输改变行为，特别是表明它不会因为传输数据宽度不同而或开或关。有一些环境下，可以通过根据传输到不同地址区域而或开或关——把系统的某部分映射为位编号一致/字节地址打乱的方式——但是你自己要保证这样能工作。

一个字节通路交换器确实达到了这样一个目标，不管你的 CPU 是设置成大尾端还是小尾端，都能够保证 CPU 和错配的外部总线或设备之间依旧保持了字节序。

正常情况下你不会把字节通路交换器放在 CPU 和局部存储器之间——这也是因为，CPU/局部存储器之间的连接又快又宽，使得字节交换器的成本太高。

如我们上面提到的，只要你能够成功译码 CPU 的系统接口，就可以把 CPU/局部存储器当成一个单元，并在 CPU/存储器单元和系统的其它部分之间安装字节交换器。在这种情形下，位编号和局部存储器内的字节顺序的关系随着 CPU 变化，但是这对系统的其它部分不可见。

对于尾端问题的错误做法和错误认识

每一个设计团队第一次碰到尾端问题时都会经过这个阶段：认为这些问题反映的是一个需要解决的硬件缺陷。情况远非那么简单。下面是几个例子：

- 可配置的 I/O 控制器：有些新的 I/O 设备和系统控制器本身可以配置成大尾端或小尾端模式。在使用这一类特性之前，你要极为仔细地阅读手册，特别是在你不是把它用作一个静态选项（设计时）而是用作一个跳线（复位时）选项的时候。

  很常见的一种情况是，这样的特性仅仅影响批量数据传输，而让程序员来处理其它的尾端问题，比如访问位编码的设备寄存器或者共享存储器的控制域。同时控制器设计人员也许并没有读过本书——关于尾端的种种似是而非的错误认识流传很广。

- 根据传输类型进行字节交换的硬件：如果你在某些字节交换的硬件上进行设计，试图解决整个问题看上去很有吸引力。如果我们只是交换字节数据以保持其地址不变，但是对于字数据则置之不理，难道不能防止整个软件问题吗？答案是不能！靠硬件修正不能解决软件问题。例如，一个实际系统中的许多传输是以数据高速缓存行作为单位的。它们可能包含不同大小和对齐数据的任意混合；如果你考虑一会儿，就会明白没有什么办法能够知道数据的边界在哪里，也就意味着没有办法确定所需的字节交换配置。

  条件字节交换仅仅是加剧了混乱，无异于火上浇油。超出无条件字节通路交换之外的任何做法都是江湖上用来骗人的玩意。

10.2.4 MIPS CPU 的双尾端软件

你可能想创建能够正确运行于任何尾端的 MIPS CPU 上的二进制代码——也许是因为要运行在可配置为任一种尾端的某个特殊板子上，或者创建一个能运行于任一种配置的板子上的可移植的设备驱动程序。这个可有点难度，很可能只要自举代码的一小部分能这样就行了。本节给出一些指南。

改变 MIPS CPU 的尾端并不需要做太多。指令集中唯一能够认识不足 32 位对象的部分就是部分字加载和存储。在一个拥有 32 位总线的 MIPS CPU 上，指令：

```
lbu t0, 1(zero)
```

获取位于程序字节地址为 1 的字节，把它加载到寄存器 **t0** 的最低有效位（0–7 位），把寄存器的其余位都填 0。这个描述是尾端无关的。但是在大尾端模式下，装进寄存器的数据将取自 CPU 数据总线的 16–23 位；在小尾端模式，字节从 CPU 总线的 8–15 位加载。

在 MIPS CPU 里头，有一个数据领航硬件，CPU 用来处理把每次传输中所有的活动字节从各自在接口上的字节通道，指引到内部寄存器的正确位置。这个领航逻辑不得不适应加载数据宽度、地址和对齐方式（包括第 8.5.1 节讲述的 load/store left/right 指令）的全部排列组合。

正是活动字节通路和部分字存取的地址之间的关系刻画了 MIPS CPU 的尾端。当你重新配置 MIPS CPU 的尾端时，是数据和寄存器之间的领航逻辑的行为发生了变化。

与芯片的尾端可配置性互补的是，大多数 MIPS 工具链可以根据命令行的选项，产生任意尾端的代码。

如果你在一个系统中设置错了 MIPS CPU 的尾端，那么会发生几种情况。

首先，如果你别的东西什么都没有变，软件很快就会崩溃，因为对任何的部分字写操作，存储器系统将会从 CPU 总线的错误部分接收到垃圾数据。在重新配置 CPU 的同时，我们最好重新配置译码 CPU 周期的逻辑。[2]

如果你修正了这一点，就会发现和系统的其余部分相比，CPU 看到的字节地址完全乱套了。用上面的话来说，我们隐含地选择了一个保持位号一致而不是字节地址的连线方式。

当然了，CPU 改变尾端后写入的数据对 CPU 自己来说没问题；如果我们只允许在复位的时候才能改变尾端，那么属于 CPU 的私有的易失存储器就不会带来麻烦。

还要注意到 CPU 看到的在对齐的总线宽度字内的位编号仍然和系统其余部分保持一致。这就是我们早些时候讲的位号一致，我们当时建议你一般应该避免。但在这个具体的例子中，这样做有一个有用的副作用，因为 MIPS 指令编码为 32 位字内的位域。对于大尾端 CPU 有意义的指令 ROM 对于小尾端 CPU 也有意义。允许我们共享自举程序代码。当然没有完美的事情——这种情况下，ROM 中任何不是由恰好对齐的 32 位字构成的数据都会乱套。许多年以前，Algorithmics 的 MIPS 板子在引导 ROM 中仅有足够的空间放下双尾端代码，来监测主 ROM 程序和 CPU 的尾端是否匹配并打印出帮助信息：

```
Emergency - wrong endianness configured.
```

`Emergency` 这个单词作为一个 C 的字符串保存，以空字符结尾。你现在应该有足够知识能理解为什么 ROM 启动代码中包含如下谜一般的程序行：

```
.align  4
.ascii  "remEcneg\000\000\000y"
```

[2]有些 CPU 接口的部分字传输采用独立的字节通道选通信号，就不会发生这种问题。

	31 … 24	23 … 16	15 … 8	7 … 0
	r	e	m	E
Byte address from BE CPU	0	1	2	3
Byte address from LE CPU	3	2	1	0
	c	n	e	g
Byte address from BE CPU	4	5	6	7
Byte address from LE CPU	7	6	5	4
	x	x	\000	y
Byte address from BE CPU	8	9	10	11
Byte address from LE CPU	11	10	9	8

图 10.8: 混合模式时打乱的字符串；参见正文

这就是字符串 `Emergecny`（带有标准 C 的结尾空字符加上两个填充字节）在尾端错误时看上去的样子。如果其开始地址不在四字节对齐的边界上，情况会更糟糕。图 10.8（从大尾端的角度来画的）演示到底怎么回事。

你看到了写出双尾端软件是可能的，但是要意识到当你准备加载进 ROM 的时候，你将要求你的工具作一些并不是为此设计的工作。典型的，大尾端工具把指令字打包为一个其中最高位在前的字节文件，小尾端工具正好想反。你要仔细考虑想要达到的结果，并检查生成的文件以确认一切按计划进行。

10.2.5 可移植性和尾端无关的代码

按照一个得到普遍遵守的约定，大多数 MIPS 工具链定义如下的 `BYTE_ORDER` 符号：

```
#if BYTE_ORDER == BIG_ENDIAN
/* big-endian version... */
#else
/* little-endian version... */
#endif
```

这样如果你真的需要，可以放一些不同的代码处理每种情况。但是最好——只要可能——还是写尾端无关的代码。特别是在充分受控的条件下（比如当给一个 CPU 可以初始化为任意一种模式的 MIPS 系统写代码的时候），认真思考一下就可以消除许多依赖性。

所有从外部数据源或者设备接收数据的引用都有潜在的尾端依赖问题。但是根据系统的布线方式，可以生成在两种尾端下都能运行的代码。在不同尾端之间接线只有两种方式：一种保持字节地址不变，另一种保持位号不变。在系统特

定区域内访问具体的外设寄存器，其尾端的变化极有可能与二者之一保持一致。

如果你设备的典型映射保持字节地址兼容，那么你应当严格按照字节操作编程。如果为了效率或者出于不得已，想要一次传送多个字节，你需要编写根据尾端条件进行打包和解包数据的代码。

如果你的设备是在字（32 位）的层次上兼容——例如，由连线到 MIPS 数据总线位固定部分的寄存器构成——那么编程时以总线宽度进行读写操作。那就是 32 位或 64 位的读写操作。如果设备寄存器不是接到 MIPS 数据总线从 0 开始的位，你可能在读之后和写之前要对数据进行移位。例如，接到一开始设计为大尾端的系统的 32 位总线上的 8 位寄存器，通常连线到位 31–24。

10.2.6 尾端和外来数据

本章是关于编程的，而不是讲 I/O 和通信的，所以本节很短。任何不是在你的代码、选用的库和操作系统中初始化过的数据一律是外部数据。可以是你从某个存储器映射的硬件读取的数据、由 DMA 放进内存的数据、并非你的程序一部分的预先编程进 ROM 的数据、或者你正在试图解释从你的操作系统下的一个“抽象”的 I/O 设备获得的字节流。

第一步是找出这个数据在内存中看上去是什么样子；在 C 语言里通常可以通过把数据内容作为一个 `unsigned char` 的字符串映射出来。即使你对数据和编译器有足够了解，能够猜出哪种 C 语言结构成功映射到该数据，当事情不符合期望的时候还是要重新回到字节数组上来；如果你的数据结构不正确，就太容易错过问题的真相了。

除了尾端之外，数据可能由你的 CPU /编译器不支持的数据类型构成；可能有类似的数据类型但是编码完全不同；可能有熟悉的数据但没有正确对齐；或者落入了本节的话题范围内，即拥有错误的尾端。

如果数据在到达你的通路链条上的每个环节都保持了字节序，可能发生的最坏情形就是整数数据以相反的次序表示，很容易构造一个“swap”宏来恢复二、四、八字节的整数值。

但是如果数据途经一个位号一致/字节地址打乱的接口，那就难办了。在这种情况下，你先要定位发生数据字节交换所在的总线宽度的相应边界；然后抽出这些边界内的字节组，不管三七二十一就进行字节交换。如果没有出错，现在就应该得正确字节顺序的有意义结果了，当然你还要处理数据中的通常问题——包括可能需要再次交换多字节整数数据。

10.3 高速缓存可见性带来的问题

在第 4.6 节，你学会了可用来正确初始化和运行高速缓存的操作。本节警告你注意可能出现的一些问题，并解释你怎样做可以解决问题。

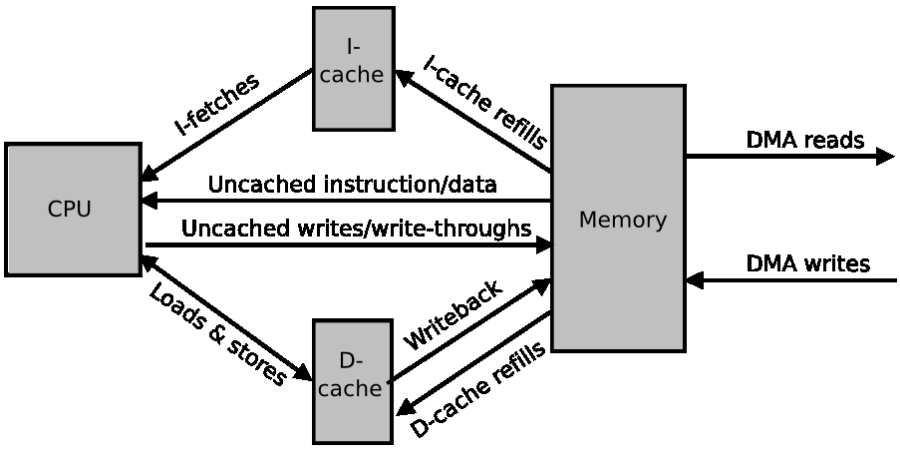

图 10.9: CPU、存储器和高速缓存间的数据流

大多数的时候，高速缓存对于软件是完全不可见的，只是加快系统的速度。但特殊情况下，如果你想要处理 DMA 控制器及类似设备，把高速缓存想象成独立的缓冲存储器可能更有帮助，如图 10.9 所示。[3]

要记住重要的一点，在高速缓存和内存之间的传输总是以适合高速缓存行结构的内存块——典型情况为 16 或 32 字节对齐的块——为单位，所以高速缓存可能因为 CPU 仅仅直接引用其中包含的一个字节而读写整个块；即使 CPU 从来没有碰过同一个高速缓存行的其它字节，仍然会在下一次传输该高速缓存行的时候包括所有的字节。

在一个理想的系统中，我们总能肯定内存的状态的更新能跟上 CPU 的所有操作请求，每个有效的高速缓存行包括相应内存位置的精确拷贝。不幸的是，实际的系统并非总能达到这个理想。我们假设在每次复位后你都初始化高速缓存，而且你避免（而不是试图解决）了第 4.12 节讲的可怕的高速缓存重影。从这些假定开始出发，一个真实的系统的行为和理想的有多大差别？

- 高速缓存中的过时数据：当你的 CPU 写入被高速缓存的存储器空间时，它更新高速缓存中的拷贝（也可以同时写入内存）。但是如果某个存储器位置的值被用别的方式做了更新，那么高速缓存的内容继续保持老的值，现在就过时了。这在 DMA 控制器写数据的时候就会发生。或者，当 CPU 为自己写了一条新指令，指令高速缓存可能继续保持该内存位置上以前的值。程序员要认识到硬件通常不能自动处理这些条件，这一点很重要。

- 存储器中的过时数据：当 CPU 写一些数据到一个（回写式）高速缓存行的时候，该数据并不立即拷贝到存储器。如果后来 CPU 读取数据，就取得高速缓存中的数据所以没有问题；但是如果是 CPU 以外的单元读取存储器，

[3] 对于具有简单的透写数据高速缓存的 MIPS CPU，图中标为“write-back”的部分并不存在。

可能就会得到老的数值。这可以发生在向外传送数据的 DMA 传输上。

用来对付因为高速缓存可见性导致的问题的软件武器中，有几个是基于 MIPS 的 **cache** 指令之上标准子程序，能让你理清高速缓存/内存的一致性问题。它们在对应于一个给定内存区域上的高速缓存位置上操作，可以写回最新的高速缓存数据或者作废高速缓存的数据（或者二者都干）。

当然了，你总可以把数据映射为不经高速缓存访问。实际上，有些环境下，只要这样做就行了：例如有些网络设备控制器，有个驻留内存的控制结构在那里读写字节和位标志，如果你把这个控制结构映射为不经过高速缓存的话要容易很多。当然对于存储器映射的 I/O 寄存器，需要对读了什么写了什么做完全的控制，也同样适用。你可以通过 kseg1 或者其它非高速缓存的空间的指针访问这些寄存器；如果用高速缓存的空间访问 I/O，肯定会坏事的。

如果（特殊情况下）你需要用 TLB 映射硬件寄存器访问，可以把页面标记为不经高速缓存。如果有人制造了一个 I/O 寄存器不在低 512M 的物理存储器空间的硬件，那么这个办法就派上用场了。

也有这种可能，你想要把一个类似存储器的设备（可能是一个图形帧缓冲区）经过高速缓存空间进行映射，这样可以从 CPU 用来实现高速缓存重填和回写的块读写的速度中受益。但是你不得不在每次访问的时候都用作废或回写来直接管理高速缓存。有些嵌入式 CPU 提供了奇妙的高速缓存选项可用于这类的硬件——检查你的手册。

10.3.1 高速缓存管理和 DMA 数据

这是常见的错误源，最有经验的程序员有时也会在这里出错。不要为此太担心：只要你仔细的想清楚了要达到的目标，就能让高速缓存的行为如你所愿，同时仍然能够让 DMA 传输既平滑又高效。

当 DMA 设备把数据放进内存的时候，比如收到了网络数据，大多数 MIPS 系统并不更新高速缓存——即使有些高速缓存行当前存放的地址位于 DMA 传输刚刚更新过的区域内。如果 CPU 随后读取这些高速缓存行的信息，就会得到高速缓存中老的过时的版本；就 CPU 的角度来说，数据依然标记为有效，没有什么东西表明内存中有了更新的版本。

为了避免这点，你的软件必须在 CPU 有机会再次引用这些数据之前，主动作废任何落入 DMA 缓冲区范围内的地址的高速缓存行。如果所有的 DMA 缓冲区的起始结束地址都正好落在高速缓存行的边界，那么管理起来就会容易得多。

对于向外的传输，在允许 DMA 设备从内存传输数据之前——比如通过网络接口发送出去的包——你必须绝对保证要发送的数据没有仅存于高速缓存的。在软件完成写入有关 DMA 传输的数据信息之后，必须保证对于任何高速缓存行，凡是当前存储的地址位于 DMA 控制器要传输的地址范围内的，一律要强制回写到内存。只有在这时，你才可以安全地发起 DMA 传输。

在一些 MIPS CPU 上，通过把高速缓存的行为配置为使用透写而不是回写，就可以避免直接进行回写操作了，但这个办法得不偿失——透写总体上要慢很多，而且会增加系统的功耗。

通过从非高速缓存地址区域访问所有 DMA 传输使用的内存，你可以真正去掉直接的高速缓存作废和回写操作。这种做法也不推荐，因为几乎肯定会把整体性能降到远低于你想要的水平。即使你的软件对缓冲区的访问是完全按顺序的，对 DMA 缓冲区进行高速缓存意味着信息的读写以高效的高速缓存行的突发方式而不是单次传输。最佳的普适的建议是对一切都进行高速缓存，但是下面的情形例外：

- I/O 设备寄存器：可能显而易见，但还是值得指出。MIPS 没有专用的输入输出指令，所有的设备寄存器都必须映射到地址空间的某个地方，如果不小心被高速缓存，就会发生很奇怪的事情。

- DMA 描述符数组：复杂的 DMA 控制器用驻留于内存的描述符数据结构和 CPU 共享控制/状态信息。典型做法是，CPU 用这些信息创建一个长长的待传输信息的列表，然后才告诉 DMA 控制器开始工作。如果你的系统使用描述符结构，你希望访问存该结构所在的内存区域的时候不经过高速缓存。

一个象 Linux 那样可移植的操作系统必须处理很大范围内各种不同类型的高速缓存，从最复杂的透明高速缓存到极为简单原始的类型，因而提供了一个明确定义的 API（稳定的功能调用集合）给设备驱动程序作者使用，还有一些介绍怎样使用 API 的简明文档。参见第 15.1.1 节。

10.3.2 高速缓存管理和向内存写入指令：自修改代码

如果你的代码要先向内存写入指令，然后执行这些指令，你一定要确保考虑了高速缓存的行为。

这可以在两个层次上让你感到惊讶。首先，如果你有一个回写的数据高速缓存 D-cache，你的程序写出的指令可能还没有到达主存，直到有什么东西触发相应高速缓存行的一次回写之前都是如此。你试图执行你的程序写出的指令的时候，这些指令还只是停留在 D-cache 内，CPU 取指不能访问数据高速缓存。所以第一步就是对相应的写入指令的高速缓存行执行回写操作；那样至少可以保证指令到达主存。

第二个惊讶的（不管你用的那种类型的 D-cache）是即使写出新指令到达了主存的某个区域，你的 CPU 的 I-cache 可能依然放的是在这些地址上过去存放的信息的拷贝。在你告诉 CPU 执行新写入的指令之前，软件首先要作废包含受影响地址范围内信息的所有 I-cache 高速缓存行。

当然了，你可以通过在非高速缓存的地址区域写入和执行新指令，以避免这些直接的回写和作废操作；但是那样做放弃了高速缓存的优势，几乎总是一个错

误。

第 4.9 节描述的通用的高速缓存管理指令都是 CP0 指令，只有核心特权级软件才能使用。当高速缓存操作和 DMA 操作有关的时候这点无所谓，因为都完全是核心的事情。但是对于先写入指令后执行的应用程序（想象一下现代的使用“及时” 解释/翻译的语言）来说就有影响了。

所以 MIPS32/64 提供了 **synci** 指令，对你的新代码中一个高速缓存行大小的代码块，一次操作内完成 D-cache 回写和 I-cache 作废。具体怎么做，参见第 8.5.11 节。

10.3.3 高速缓存管理和非高速缓存或透写的数据

如果你把映射到同一个物理地址范围内的高速缓存和非高速缓存访问混合起来，就要考虑一下这样做对高速缓存意味着什么。非高速缓存的写仅仅更新给定地址在主存中的拷贝，可能让 D-cache 或 I-cache 中那个位置上的内容变成过时的拷贝。非高速缓存的加载将会选择在主存找到的内容——即使那个内容相对于高速缓存中目前的拷贝已经过期。

在复位后将你的系统带入到已知状态的底层代码中，对同一个物理地址小心使用高速缓存和非高速缓存的访问有一定用处，有时甚至是必要的。但是对于运行的代码，你可能不想这样做。对于每一块物理存储器区域，决定你的软件访问到底是经高速缓存还是不经高速缓存，一旦决定访问方式此后就要绝对保持一致。

10.3.4 高速缓存重影和页面着色

在第 4.12 节对于高速缓存重影问题的硬件起源讲得更多。问题出现于 L1 一级高速缓存采用了虚拟索引和物理标签，而且索引的范围大到占用了两个或者更多页的大小。索引的范围为高速缓存每一“组”的大小，那么采用常规的 4K 大小的页，在 8K 的直接映射高速缓存或者 32 K 的四路组相联高速缓存中就会出现重影。

一个位置的“页颜色”就是那些用来在每路高速缓存组中选择页大小的存储块的一个或者多个虚拟地址位。指向同一块物理数据的两个虚拟指针仅当拥有不同的页色时才有可能产生重影。只要指向同一数据的所有指针所处的页都拥有相同 的颜色,那么一切都很好——任意数据，即使位于多个不同的虚拟地址，也会被存储到高速缓存的同一个物理部分，由同样的物理标签来标识。

一个物理页可以从多个虚拟页面位置访问（共享库就常常在拥有不同虚拟地址的程序之间共享），这在 Linux 中司空见惯。

大多数时间，操作系统对于共享数据的虚拟地址的对齐要求都会更高——共享的进程不一定用同样的地址，但是我们要确保不同的虚拟地址之间相隔开的距离都为，比如说 64 KB 的整数倍，那么不同的虚拟地址都是同

样的页色。那样多花了一点虚拟存储器空间，但是虚拟空间相当便宜。

很容易会想到只要数据是“只读”的（当然必须至少要写一次，但那在重影发生之前了）高速缓存重影就是无害的：我们不在乎一个只读页有多个拷贝。但是只能说大多数时间是无害的。对只读数据的重影或许可以容忍，特别是在 I-cache 中：但是你要保证高速缓存管理软件知道在一个虚拟地址被作废的数据有可能还在高速缓存的另一个位置上。

随着虚拟存储操作系统（特别是 Linux）在嵌入式和消费类计算市场的广泛使用，越来越多的 MIPS CPU 在制造时就保证不会发生高速缓存重影。修正这个长期以来的缺陷只是一个时间问题。

不管你要做什么，MIPS32/64 CPU 必备的基本高速缓存操作，在第 4.9.1 节都讲到了。

10.4 访问内存的次序安排及调整

程序员倾向于认为他们的代码是按照良好顺序执行的：CPU 看一下指令，用适当的方式更新系统的状态，然后继续下一条指令。但是如果允许 CPU 打破这个纯粹的串行执行方式，使得操作不一定局限于按照严格的程序顺序执行，那么程序就可以运行得更快。这对于在处理器接口处因执行存取指令而触发的读写事务尤其重要。

从 CPU 的角度看来，一次存储只需要一个对外的写请求：给出内存地址和数据，交给存储器控制器完成工作。实际的存储器和 I/O 设备相对较慢，在写操作执行的过程中，CPU 可以运行几十条甚至几百条指令。

当然读操作就不同了：它需要双向的通信，一个对外的请求和一个对内的响应。当 CPU 需要知道一个内存地址的内容或者设备寄存器的内容时，在系统给出其响应信息之前恐怕做不了多少事情。

要追求更高的性能，就意味着我们要让读操作尽可能的快，甚至不惜让写操作变得更慢。把这个思想进一步引申，我们就可以让写操作排队等待，把任何随后的读操作请求提前传递到缓冲写的前面。从 CPU 的角度看来，这是个大优点；通过立即开始读取事务，能够尽快得到响应。写操作也得找个时间来做，因为队列长度是有限的：但有可能是在这次读操作完成之后 CPU 要在高速缓存中执行一段时间。如果写缓冲队列已经满了，我们只需要在写操作的时候停下来等一会儿：这当然不会比我们按顺序执行写操作的效果差。

你可能看出这里有一个问题：有些程序写完内存然后又读回来。如果读操作抢先于写，那么我们可能从内存中获得过时的数据，这样我们的程序就无法正常工作。大多数时间里我们可以加上额外硬件，对照检查读操作请求和写队列中的地址，并在二者匹配时不允读操作许抢先于写操作。通过这样一个额外硬

件就可以修正这个问题。[4]

在那些真正并发（即运行于不同的 CPU 上）的任务间共享变量的系统中，读写的顺序问题变得更加危险。不错，在多数时间任务之间没有互相次序的问题。当任务有意用共享内存变量进行同步和通信的时候，次序确实有影响；但在这种情况下，软件将会用精心设计的操作系统同步操作（比如锁和信号量）。

但是共享内存有一些技巧——往往是好用、廉价、有效的技巧——不需要用那么多的信号量或者锁，但是容易被执行次序的随意变化破坏。例如假设我们有两个任务：一个任务正在写某个数据结构，另一个正在读同一个数据结构。二者轮番使用该数据结构，如图 10.10 所示。

要操作正确，我们需要知道当读任务看到关键字域值已经更新时，能够保证所有其它的更新也会对读任务可见。

除非我们抛弃解除 CPU 的读写之间的耦合所带来的全部性能优势，让硬件对程序员隐藏所有的次序问题是不现实的。MIPS 体系结构提供了 **sync** 指令用于这个目的：可以向你保证（对于共享内存的所有参与者）在 **sync** 指令之前出现的所有读写操作都会在其后出现的操作之前完成。值得仔细思考一下这一承诺的局限性：仅仅限于次序，而且仅限于参与对非高速缓存或者一致性高速缓存的存储器访问的任务可见的效果。

要让以上的例子在合适的系统上可靠运行，写任务应该在写 `keyfield` 之前包括一条 **sync**，读任务在读 `keyfield` 之后应当要有一条 **sync** 指令。详情参见第 8.5.9 节。但是关于这个话题还有很多东西；如果你正在构建这么一个系统，强烈建议你用一个提供了适当的同步机制的操作系统，并就这个题目作进一步阅读。

不同的体系结构对于执行顺序做出了不同的承诺。一种极端是，要求所有的 CPU 和系统设计人员努力保证一个 CPU 的全部写和读操作从另一个 CPU 的角度看上去次序完全相同：那叫做“强有序。”也有些弱一点的承诺（比如“全部写操作要保持次序不变”）；但是 MIPS 体系结构采取了极为激进的立场，即根本不做任何保证。

10.4.1 访存次序和写缓冲器

让我们离开玄虚的理论来讲点实际的东西吧。把对外的请求保存在一个“写缓冲器”的思想在实践中证明非常有效，因为存储指令倾向于扎堆出现在一起。对于一个运行着编译出的 MIPS 代码的 CPU，典型的结果表明只有约百分之十的指令是存储指令；但是这些访问倾向于突发进行——例如，在函数首部保存一组寄存器的值的时候。

大多数时候，写缓冲器的操作对于软件完全透明。但在有些特殊的情况下，程序员需要知道发生了什么情况：

[4] 你可以，在某些情况下，从写缓冲器返回数据来满足读请求。但是若我们允许读操作先等待一下然后访问内存的话，就不需要发明任何东西。

写任务	读任务

...
/* update entries */
keyfield = WRITEDONE;
sendsignaltoreader();
	keyfield = WAITINGFORWRITE;
	...
	
	while (keyfield != WRITEDONE) {
	waitforsignalfromwriter();
	}

图 10.10: 共享数据结构的任务

- I/O 寄存器访问的时序关系：这会影响到所有的 CPU。在 CPU 执行一次存储操作更新 I/O 设备寄存器时，对外的写请求在到达设备的路上，有可能在写缓冲器受到一些延迟。其它事件，比如对内的中断，可能发生于 CPU 执行存储指令之后，但在写请求于 I/O 设备中生效之前。这可以导致奇怪的行为：例如，在你告诉一个设备不要中断之后，CPU 还可能收到来自该设备的中断。再给一个例子：如果一个 I/O 设备需要软件实现的延迟以便从某个写操作恢复，就必须保证在你开始对延迟计时之前写缓冲器为空——也要保证在写缓冲器清空期间 CPU 要等待。好的做法是定义一个子程序完成这个工作，习惯上该例程命名为 `wbflush()`。具体实现参见第 10.4.2 节。

- 读抢先于写：MIPS32/64 体系结构允许上面讨论过的这种行为。如果你的软件要健壮并且要易于移植，就不应该假定会保持读写的顺序。当你需要保证两个周期按某个特定次序发生时，你需要第 8.5.9 节讲述的 **sync** 指令。

- 字节收集：有些写缓冲器观察同一个内存字中的部分字写操作（乃至同一个高速缓存行内的写操作）把这些部分写操作合成为单个的写操作。

  为了避免将非高速缓存区的写操作合并成一个字宽度的操作而带来的不良反应，一个好办法就是把你的 I/O 寄存器映射到让每个寄存器分别位于单独的字地址（比如：8 位的寄存器至少要离开四个字节）。

10.4.2 实现 wbflush

大多数写队列可以通过执行一个到任意位置的非高速缓存的存储然后接着读取同一数据的操作来清空。一个写队列当然不能允许读抢先于写——那样会

返回过时的数据。在写和读之间放一个 **sync** 指令，这对任何符合 MIPS32/64 标准的系统应当都有效。

这样做可行，但是不一定效率高；通过从最快的可用存储器加载数据可以把开销减到最小。也许你的系统提供了某些系统特有的但更快的机制。使用前请先阅读下面的注意事项!

注意! 写缓冲器常常在 CPU 内部实现，但是也可能在外部实现；任何声称具有 *write-posting* 特性的系统控制器或者存储器控制器接口，都在系统中引入了另外一级的写缓冲。CPU 外部的写缓冲器会引起和内部的写缓冲器完全相同的麻烦。需要仔细找出你的系统中的全部写缓冲器都位于什么地方，在编程的时候要考虑。

10.5 用 C 语言开发

你可能已经几乎一切都用 C 或者 C++ 来写了。MIPS 缺乏专门的 I/O 指令意味着 I/O 访问就是对适当选择的存储器地址的加载和存储；这很方便，但是对 I/O 寄存器的访问通常有一定的约束，你要保证编译器不要太聪明了。MIPS 大量使用 CP0 寄存器也意味着 OS 代码可以从使用精心挑选的 C 的 `asm()` 操作中受益。

10.5.1 用 GNU C 编译器包裹汇编代码

GNU C 编译器（“GCC”）允许你在 C 语言源文件中嵌入汇编代码片断。GCC 拥有一些特别强大的特性，其它现代的编译器可能也支持这里的例子，但是语法可以有很大不同，所以我们这里只讨论 GCC。

如果你对硬件的底层控制需要不止几条机器指令，比如需要一个库函数来执行某些复杂的计算，那你真正要理解怎样写纯粹的汇编程序；但是如果你只是想插入一条或几条特殊的 MIPS 指令组成的短序列，**asm()** 指示语句可以相当简单的达到想要的效果。更好的是，你仍然可以把管理寄存器选择的任务交给编译器，让其按照自身的约定去处理。

作为一个例子，下面的代码让 GCC 使用新近的 MIPS CPU 上都有的三操作数形式的乘法。如果只是使用正常的 C 语言的 `*` 乘法运算符，这个工作最后可能使用原先的乘法指令的形式，即有两个源操作数，隐含地把双倍宽度的结果发送到 **hi/lo** 寄存器对。[5]

C 函数 `mymul()` 和三操作数的 **mul** 指令行为完全相同，把双倍长度结果的低一半的有效位返回；高一半就被简单丢弃了。由你自己保证要么不会溢出，要

[5]在本书写作的时候，GCC 有一个使用 MIPS32 的三操作数 **mul** 指令的版本已经在流传了——但这仍然是一个很好的例子。

么溢出也没关系。

```
static int __inline__ mymul(int a, int b)
{
    int p;

    asm("mul %0, %1, %2"
        :"=r"(p)
        :"r"(a), "r"(b)
    );

    return p;
}
```

函数本身被声明为 `inline`，这告诉编译器对这个函数的调用应当用其函数体（允许施加局部寄存器优化）来替换。加上一个 **static** 表示该函数不需要公开给其它模块使用，所以并不生成函数自身的二进制代码。这样包裹 `asm()` 往往是有意义的：你通常可以把整个定义放进一个 include 文件。你也可以使用一个 C 预处理器的宏，但是内联函数略为清晰。

`asm()` 括号里面的声明告诉 GCC 生成一条 MIPS **mul** 指令行给汇编器，该行带三个操作数——一个输出，两个输入。

在下面一行，我们告诉 GCC 乘积操作数 `%0` 的要求：首先，用“`=`” 修饰符表示这个值是只写的（就是说没有必要保留它先前的值）；“`r`”告诉 GCC 可以自由选择任何一个通用寄存器来保存该值。最后我们告诉 GCC 我们写成 `%0` 的操作数对应于 C 的变量 `p`。

在 `asm()` 结构的第三行上，我们告诉 GCC 有关操作数 `%1` 和 `%2` 的情况。我们允许 GCC 把它们放到任何通用寄存器中，并说它们对应于 C 的变量 `a` 和 `b`。

在本例中函数的末尾，我们把从乘法指令得到的结果返回给 C 调用程序。

GCC 在指定操作数的时候提供大量的控制，你可以指定一些值是同时可读可写的，某些硬件寄存器会因为特定的汇编序列的副作用而导致无意义的值。你可以从 GCC 手册有关 MIPS 的部分挖出更多细节。

10.5.2 存储器映射的 I/O 寄存器和“volatile”

大多数读者会用 C 写访问 I/O 寄存器的代码——你当然不应该用汇编代码，除非有什么极为迫切的要求，而且因为 MIPS I/O 寄存器都是存储器映射的，从 C 语言中访问并不难。说了这么多，你心里要记住一条，随着编译器发展或者你大量使用了 C++，要准确地预测代码最后产生的底层指令序列变得更加困难。这里给出一些经过长期实践检验的提示。

我要写一段代码，想要检测一个串口的状态寄存器，如果状态是就绪就发送一个字符：

```
unsigned char *usart_sr = (unsigned char *) 0xBFF00000;
unsigned char *usart_data = (unsigned char *) 0xBFF20000;
#define TX_RDY 0x40

void putc(char ch)
{
    while ((*usart_sr & TX_RDY) == 0)
        ;
    *usart_data = ch;
}
```

要是这个程序发送了两个字符然后就死循环，我肯定会感到心烦意乱，但很有可能就会发生这种情况。编译器看到 `*usart\_sr` 对应的存储器映射的 I/O 的引用是一个循环不变量；在 `while` 循环中没有修改它的操作，所以看上去似乎把加载操作提到循环外面是安全的。编译器认为你的 C 程序等价于：

```
void putc(char ch)
{
    tmp = (*usart_sr & TX_RDY);

    while (tmp)
        ;
    *usart_data = ch;
}
```

这个具体问题，可以通过下面的方式定义寄存器来防止：

```
volatile unsigned char *usart_sr =
    (unsigned char *) 0xBFF00000;
volatile unsigned char *usart_data =
    (unsigned char *) 0xBFF20000;
```

如果你要检查一个被中断或者其它异常处理程序修改的变量的值，也会有类似情况。声明该变量为 `volatile` 应该能修正这个问题。

我不保证这种做法永远有效：C 的圣经里头说 `volatile` 取决于具体实现。我怀疑默认不允许忽略 `volatile` 关键字的编译器优化掉加载。

许多程序员用 `volatile` 时有困难。只要记住它和 C 语言别的类型修饰符行为一样——就像上例中的 `unsigned` 一样。你要避免象下面这样的综合症：

```
typedef char *devptr;
volatile devptr mypointer;
```

现在你告诉编译器必须坚持从变量 devptr 加载指针值，但是对于该指针指向的寄存器的行为并没有说。把代码写成下面这样更有用：

```
typedef volatile char *devptr;
devptr mypointer;
```

一旦你处理了这个之后，优化导致代码不能正确运行的最常见的原因就是处理的速度太快硬件跟不上了。硬件寄存器的读写常常有定时方面的限制，你常常不得不故意放慢代码的速度以与之相适应。

本节的主旨是什么？虽然用 C 语言书写和维护硬件驱动程序代码比汇编语言要容易一些，负责任地使用这一选择仍然很重要。特别是你需要对工具链把高级语言源程序代码转化成底层机器指令的方式有足够的了解，以保证系统的行为是你想要的。

10.5.3 用 C 语言开发 MIPS 应用时的其它杂七杂八的问题

- 负指针：当在 MIPS CPU 上运行不作地址映射的简单代码时，所有指针都位于 kseg0 和 kseg1 区域，这样任何 32 位数据指针值的最高位都是一，看上去是“负值”。不作地址映射的程序在大多数其它体系结构上都是直接处理物理地址，通常远远小于 2G!

  如果指针被隐式转换成有符号整数类型，这样的指针值在作比较的时候会引起麻烦。整数和指针类型之间的隐式转换（在 C 语言里很常见）应当修改为显式转换，而且应当转换为无符号的整数类型（这里应该用 unsigned long）。

  大多数编译器都会对指针-整数转换进行警告，尽管有时要指定一个选项才行。

- 有符号和无符号的字符：在早期的 C 编译器中，用于字符串的 char 类型通常等价于 signed char；这与更大的整数类型是一致的。但是，一旦你不得不处理超过 7 位值的字符编码，这种有符号字符在转换和比较的时候都很危险。现代的编译器通常让 char 等价于 unsigned char。

  如果发现你的老程序依赖于 char 类型缺省的符号扩展，好的编译器提供有一个选项可以恢复传统的约定。

- 对 16 位 int 的提升：还有不少的程序正从 16 位的 x86 或者其它 CPU 上转移过来，这些 CPU 上标准整数为 16 位。这样的程序有可能以一种超出想象的、极为微妙的方式，依赖于 16 位值的有限大小和溢出特性。虽然你可以把这些类型转换成 short 正确运行，但那样效率不高。大多数情况下，

你可以让变量值悄悄转为 32 位的 MIPS `int`，但是你应当特别注意那些用有符号数的比较来捕获 16 位溢出的地方。

- 依赖于栈的程序设计：有些类型的函数调用栈和数据栈隐含在 C 语言的块结构中。尽管 MIPS 硬件完全没有提供对栈的支持，MIPS C 的编译器实现了一个相当传统的栈结构。话虽如此，但是你的程序真的要是认为它知道栈的结构，就会导致无法移植。只要可能，就不要想当然地用新的假设取代旧的假设：现在有一些得到普遍认可和符合标准的宏/库操作，或许能解决以前你的软件试图自己做的事情，这样就消除了滥用栈的两个最常见的动机：

  - `stdargs`：用这个基于 include 文件的宏包来实现编译时参数个数和类型不确定的子程序。
  - `alloca()`：要在运行时分配内存，调用这个库函数，它分配的内存是“栈上”的，意思是说当申请内存的函数返回时自动释放空间。有些编译器对 `alloca()` 的实现其实是一个扩展栈的内建函数；要不就提供一个纯粹库函数的实现。但是不要想当然地就认为这样分配的空间实际上位于与栈有关系的地址上。

第 11 章　MIPS 软件标准 (ABI)

在本书的大部分，我们是从程序员的眼里看待硬件的角度来介绍体系结构。本章来介绍一下与怎样创建互相兼容的 MIPS 二进制程序有关的标准。

这些标准围绕着硬件的特性来设计，但是常常带有一定的任意性——总得用一种方式来做，要是每个工具链都用同样的方式做就会带来明显的好处。我们已经碰到过其中一个标准：第 2.2 节介绍的寄存器使用约定。

本章我们要看一下编译器在函数参数传递和使用堆栈时，怎样表示 MIPS 程序的数据。我们所有的例子全部采用 C 语言，当然本质上同样的约定也适用于其它的语言。数据表示和函数链接属于称为应用程序二进制接口 ABI (Application Binary Interface) 的正式标准的一些方面。我们介绍的是 ABI 中有时叫做 o32 的使用约定。但是 ABI 文档还规定了目标文件（用来容纳二进制程序和库的文件格式，比如 ELF 等）的编码，本书不涉及这部分。

Linux 还要有更多的约定，才能让各部分程序和共享库动态链接到一起构成一个应用程序。我们将在第 16 章讲这个问题。

数据的组织受到我们在第 10.2 节详细讲述的 CPU 尾端的深刻影响。

MIPS 历史上最重要的 ABI 有：

- o32: 源于传统的 MIPS 约定（“o”代表 old），这是我们这里要详细讲解的。o32 基本上依然是嵌入式工具链和 32 位 Linux 通用的约定。

- n64: 新的用于运行着 IRIX 操作系统的 SGI 64 位 CPU 上的 64 位程序的官方 ABI。SGI 的 64 位模型中指针和 C 语言的 `long` 类型都为 64 位。但是 n64 还改变了寄存器使用的约定和参数传递的规则。因为将更多的参数用寄存器传递，性能有略微提高。

- n32: 伴随 n64 的 ABI，实际上是用于 64 位 CPU 上的“32 位”程序。除了指针和 C 的 long 数据类型实现为 32 位以外，基本上和 n64 相同。这一点挺有用处——对于 32 位地址空间已经够用的那些应用程序来说，64 位的指针除了增加开销外不能提供任何好处。

表 11.1: 数据类型以及在内存中的表示

C 类型	汇编名字	宽度（字节数）
char	**byte**	1
short	**half**	2
int	**word**	4
long long	**dword**	8
float	**word**	4
double	**dword**	8

11.1 数据表示和对齐

当你在 C 语言中定义数据的时候，数据在内存中的布局依赖于编译的目标机器。[1]。而且，虽然假定有某个特定的内存布局是“不可移植的，”但是相比于其它可用的方法，不可移植的 C 在定义固定的数据布局时更易于维护。

能选择的数据布局受到硬件能力的制约。MIPS CPU 只能加载自然对齐的多字节数据——从四字节边界加载四字节数据等等——但是许多 CISC 体系结构就没有这一限制。MIPS 编译器试图保证数据位于适当对齐的地方；这就会影响到许多方面的深层次的（并不总是明显的）行为。

就本节的内容而言，存储器完全可以看作一个无符号的 8 位量的数组，其索引就是虚拟地址。对于已知的所有 MIPS 体系结构的 CPU，这个相当于于 C 语言的定义 `unsigned char []`。

和我所知道的现代计算机一样，MIPS 用二进制补码形式表示有符号整数——所以不管哪个数据宽度，-1 的二进制表示都为全一。补码表示的主要好处就是基本的算术运算（加减乘除）对于有符号和无符号数据类型的实现是一样的[2]。

C 的整数类型有 `signed` 和 `unsigned` 两个版本，二者的数据宽度和对齐要求永远相同。当你没有明确指定哪一个的时候，通常 `int`、`long`、`long long` 为 `signed`，但 `char` 为 `unsigned`。[3]

11.1.1 基本数据类型的宽度

表 11.1 列出了基本的 C 数据类型以及它们在 MIPS 体系结构的 CPU 上的实现。我们随后回头再讲 `long` 和指针类型——其宽度根据使用的 ABI 而不同。

汇编器并不区分整数和浮点数据类型的存储定义。

[1] 严格来讲，也依赖于编译器，但是实际的 MIPS 编译器都遵守本节描述的约定。

[2] 至少在结果不超出操作数精度时如此。

[3] 这是 ANSI C 的特性。早期的 C 的 `char` 缺省也是 `signed`。大多数编译器允许通过命令行选项改变 `char` 的缺省类型——这在重新编译老软件时有用。

11.1.2 “long”和指针类型的宽度

我们在表中略去了这两种类型，因为它们的宽度在不同的 ABI 中不同。但是它们二者宽度总是相同的。

对于 o32 和 n32（还有任何用于 32 位 CPU 上的合理的 ABI），`long` 的实现和 `int` 完全一样；对于 64 位 n64 的 ABI，`long` 的实现和 `long long` 一样。

在所有情形，指针的存储总是和 `unsigned long` 一样；MIPS 体系结构总是声称拥有简单的“平坦”地址空间。

11.1.3 对齐要求

所有这些基本的数据类型，只有当自然对齐的时候，才可以用标准的 MIPS 指令直接处理，就是说，双字节的数据要从偶数地址开始，四字节的数据要从 4 的倍数的地址开始，八字节数据要从 8 的倍数的地址开始[4]。

11.1.4 基本类型在内存中的布局和及其随尾端的变化

图 11.1 显示了每个基本数据类型在我们的字节寻址的内存中的排布；对于大尾端和小尾端软件的排布不同。

我在两种尾端的布局中反转了各个位的编号。这对于内存寻址来说没有意义；每个字节是一个不可分割的 8 位整体。但是把位号倒过来，使得浮点数的各个域的位图解看上去更易于理解（而且更好看）。

以上每一种数据类型都是自然对齐的。

“尾端”容易引起各种问题，我们在第 10.2 节有详细讨论。

11.1.5 结构体和数组类型的内存布局和对齐

复杂的数据类型是通过把简单数据类型拼接起来，但在其间插入无用的（“填充”）字节以满足对齐规则[5]来构成的。

这点值得举几个例子瞧瞧。下例给出 `struct mixed` 中数据项的字节偏移量：

```
struct mixed {
    char c; /* byte 0 */
    /* byte 1-7 are ‘‘padding’’ */
    double d;   /* byte 8-15 */
    short s;    /* byte 16-17 */
```

[4]对于只用 32 位寄存器和数据通路的 MIPS32 CPU 来说，没有处理八字节数据的机器指令，八字节对齐的限制并不是绝对必要的。但是已知的所有 API 都强制要求这一点。

[5]有些编译器系统提供了改变特定数据定义的对齐规则的机制。这允许你用 C 语言的数据定义设计更多更广的数据模式，并且编译器会生成适当的代码（损失一些效率）来处理未对齐的基本数据类型。第 11.1.7 节对此给出了一些提示。

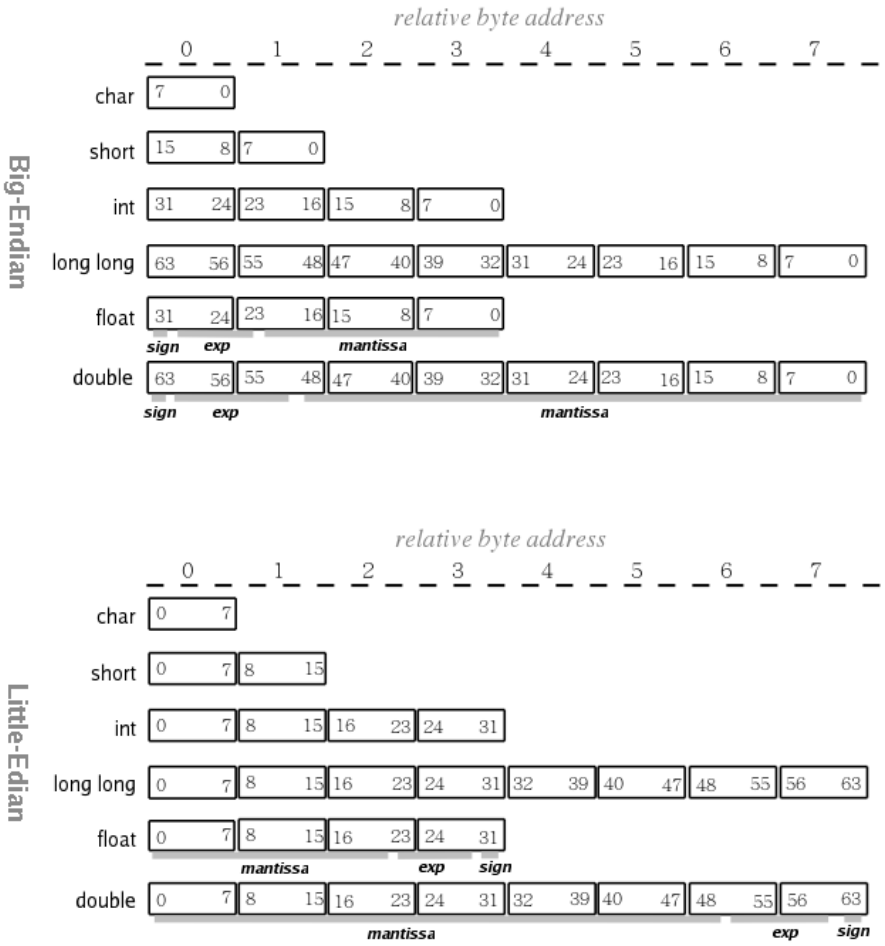

图 11.1: C 数据类型在内存中表示

```
}
```

值得强调的一点是，构造数据类型的各个域的字节偏移量不受尾端的影响。

数据结构或者数组在内存中要对齐到其所包含的数据类型所要求的最大的对齐边界。所以 `struct mixed` 从八字节边界开始；这意味着如果你用这些结构体组成一个数组，你需要在每个数组成员之间进行填充。C 编译器通过对结构体提供“结尾填充（tail padding）”使它可以用于数组，所以 `sizeof(struct mixed) == 24` 而该结构体正确的注释应该如下：

```
struct mixed {
    char c; /* byte 0 */
    /* byte 1-7 are ‘‘padding’’ */
    double d;   /* byte 8-15 */
    short s;    /* byte 16-17 */
```

```
    /* bytes 18--23 are ``tail padding'' */
}
```

只是提醒一下，指针和 long 类型的宽度和对齐要求可以为四或者八，取决于你是否用 64 位操作。

11.1.6 结构体中的位域

C 允许定义将若干个短的“位域”成员压缩进一个或多个标准整数类型的存储位置。这个特征对于仿真、硬件接口、甚至定义紧凑的数据结构也有用处，但还不止于此。位域定义名义上依赖于 CPU（其它东西又何尝不是如此），但其实也取决于尾端。

你也许还记得在第 7.9.3 节我们用位域来映射一个存放在内存中的单精度 IEEE 浮点值（C float）。一个浮点的单精度值占用多个字节，所以可以预料其定义与尾端有关。大尾端版本看上去如下：

```
struct ieee754sp_konst {
    unsigned    sign:1;
    unsigned    bexp:8;
    unsigned    mant:23;
};
```

C 的位域总是紧缩的——就是说，这些域不会被填充以满足某种对齐要求。但是编译器不允许位域跨越容纳位域的 C 类型（在上面的例子中，就是一个 unsigned，即 unsigned int 的缩写）的边界。

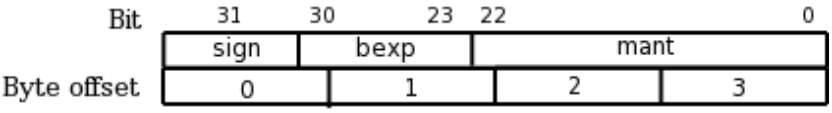

图 11.2: 大尾端视角看到的位域

大尾端 CPU 中的结构和映射如图 11.2 所示（采用了典型的大尾端图像）；小尾端版本如图 11.3 所示。

C 语言编译器坚持要求，即使对于位域，结构体中先声明的项要占据较低的内存地址：当你的系统为小尾端时，你需要把结构体的声明倒过来：

```
struct ieee754sp_konst {
    unsigned    mant:23;
    unsigned    bexp:8;
    unsigned    sign:1;
};
```

要理解为什么非得这样，从图 11.3 看出：在小尾端模式里，编译器从低编号开始把每个二进制位打包到结构体里面。

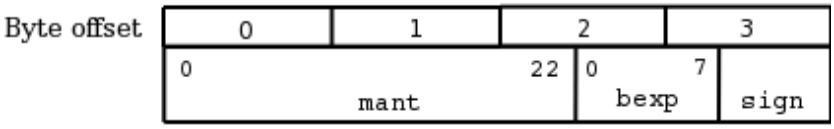

图 11.3: 小尾端视角看到的位域

这有什么关系吗？当然有了；如果你试图用对尾端依赖较少的方式实现位域时，那么在下面的例子中，`struct fourbytes` 在存储器中的分布就和 `struct fouroctets` 不同，这个结果看上去不太合理：

```
struct fourbytes {
    signed char a; signed char b; signed char c;
    signed char d;
};

struct fouroctets {
    int a:8; int b:8; int c:8; int d:8;
};
```

当察看一个浮点数内部的时候 CPU 尾端问题出现也许并不奇怪；我们前面讲过用不同的 C 数据类型存取同样的数据常常会反映出 CPU 的特性。但是这个例子也再次提醒我们尾端问题是一个真正棘手和无处不在的问题，即使没有外部数据和硬件需要管理的时候也如此。

一个域只能被存进其定义类型的一个存储单元里；如果我们要为 MIPS 的双精度浮点数定义一个结构体，尾数域包含两个 32 位存储单元的部分，所以不能在一次定义中完成。我们能做到的最好办法就是象下面这样：

```
struct ieee754dp_konst {
    unsigned sign:1;
    unsigned bexp:11;
    unsigned manthi:20;   /* cannot get 52 bits into ... */
    unsigned mantlo:32;   /* .. a regular C bitfield     */
};
```

允许在定义中省略成员域的名称，这样就不用为那些仅仅用来填充的域起名了。

如果用的是 GNU C 或者其他现代的编译器，我们用一个 `long long` 位域就能一次定义一个双精度浮点寄存器了。但是 ANSI C 的规范并没有要求编译器必须支持 `long long`。

关于位域对齐的完整规则有点复杂：

- 如上所说，位域必须完全位于适合于其声明类型的单个存储单元之中。因而位域不能跨越其单元的边界。
- 位域可以和其它结构体/联合体成员，包括非位域的成员（要存放到一起，相邻的结构体单元必须是一个较小的整数类型）共享一个存储单元。
- 结构体通常继承其成员中对齐要求最严格的成员类型的对齐要求。有名字的位域让结构体对齐到（至少）和该类型要求的同样标准。

  无名字的位域——不管其定义的类型是什么——只能强制要求存储单元或者整个结构体对齐到能够容纳那么多位的最小整数类型上。
- 你也许想强制随后的结构体成员占用一个新的存储单元。在某些编译器中，你可以用一个*无名零宽度*的位域实现这一点。其它地方零宽度的域是非法的（至少是没有意义的）。

现在你已经了解到了在能够兼容各种 ABI 的条件下，把 C 的数据声明映射到存储器所需的一切知识。

11.1.7 C 中非对齐的数据

帮助提高 MIPS CPU 运行效率的各种对齐规则有时候很烦人，因为你想要匹配另一个应用程序的数据，想要你的 C 结构体表示一个精确的字节到字节的存储器排布。

GNU C 编译器（以及其它大多数较好的编译器）允许对用来定位任意数据的对齐规则进行控制，不论数据是简单的变量或者复杂的结构体。在 GNU C 的语法中，数据声明中的`__attribute__((__packed__))` 将导致所有常规的对齐要求被忽略和整个数据的紧缩，而 `__attribute__((__aligned(16)__))` 坚持对齐到 16 字节——比 MIPS 编译器使用的任何标准数据类型所要求的对齐都更严格[6]。

适当的混合这两种属性，你可以用你喜欢的方式任意安排数据结构（单个的 `__aligned__()` 按照要求控制紧缩结构内部的对齐）。

ANSI 标准对于数据结构有个功能略少但是灵活的语法，但不适用于基本类型（可是因为它是标准，因而能够跨不同的编译器）：用一个类似的预处理命令。这样如果你包含一个以 `#pragma pack(2)` 开始的一行数据声明，编译器就会对这些数据用不超过两个字节的对齐。用 `praga pack()` 恢复正常的对齐行为。

[6] 虽然编译器会尊重你提出的大得出格的对齐要求，有时候链接器或者工具链的其它的后续部分并不如此——所以使用时要慎重。

例如下面的例子：

```
int unalignedload (void *ptr)
{
#pragma pack(1)
    /* define what you like here, with no assumptions about alignment */

    struct unaligned {
        int conts;
    } *ip;

#pragma pack()
/* back to default behavior */

    ip = (struct unaligned *) ptr;

    /* can now generate an unaligned load of int size */
    return ip->conts;
}
```

11.2 MIPS ABI 的参数传递和堆栈约定

C 语言和其它常见的编译型语言可以从分别编译的多个模块来构建程序。为了保证在模块相互之间以及和操作系统之间能够合作，模块编译出的代码依赖于有关寄存器使用、堆栈构造和参数传递等方面的（由编译器实施，因而对汇编语言程序员是强制的）统一约定。

某些读者可能没有写过 C 或者 C++ 程序。如果你用 Java 或者其它全部或部分解释的高级语言编程，那么这里讲的跟你没有什么直接关系。我写本节是针对 C 代码的（我对其它语言的了解不够多，也不知道该讲到哪里为止）。

本节的内容包括堆栈、子程序链接、参数传递以及在 MIPS 代码中怎样管理这些以便能有效满足程序员的所需的一切。有关寄存器使用的约定在第 2.2 节介绍了，有关编译器如何打包 MIPS CPU 数据的介绍可参阅第 11.1 节。

11.2.1 堆栈、子程序链接和参数传递

自从 MIPS 于 1980 年代早期出现，就有一套有关怎样给函数传递参数（“给子程序传递参数”的 C 语言说法）以及怎样从函数返回结果值的约定。

这些约定逻辑上遵循某个背后的原则：所有参数都在堆栈数据结构里分配空间，但是属于堆栈最初几个位置的内容实际上在 CPU 寄存器中传递——相应

的内存位置的值是未定义的。在实际中，这意味着对于大多数调用，参数都是在寄存器中传递的；但是栈数据结构（即使没有存入任何东西）对于理解调用过程来说是最好的切入点。

我们将详细介绍 o32 标准规定的参数传递和堆栈使用，然后在第 11.2.8 节概括一下 n32 和 n64 中的变化。

关于改进这些标准的讨论有很多：o32 非常老了，自从其发明以来编程习惯已经发生了改变。我们希望所有这些讨论能够给出一个更好的规范——但是到本书写作时（2006 年春）以及可以预见的将来，用于 32 位 MIPS CPU 上的嵌入式应用还要使用 o32 编译器，所以学习 o32 不会有多大损失。

11.2.2 o32 中的堆栈参数结构

MIPS 硬件并不直接支持堆栈，但是 C 语言的语义基本上一定要求有栈。o32 有一个向下增长的栈，当前栈底的位置保存在寄存器 **sp**（**$29** 的别名）中。任何提供保护和安全措施的操作系统不会对用户栈做任何假定，除了函数调用点之外，**sp** 的值无关紧要。但是习惯上还是把 **sp** 保存在函数使用的最低堆栈位置或者下面。

在函数调用点，**sp** 必须是八字节对齐的，对应于最大的基本类型——整数型 `long long` 或者浮点型 `double`。八字节对齐对于 32 位 MIPS 整数硬件不是必须的，但对于和 64 位寄存器的 CPU 兼容是必不可少的，因而也成为了规定的一部分。子程序总是将栈指针调整为 8 的倍数来做到这一点[7]。

按照 MIPS 的标准调用子程序时，调用者在栈上创建一个容纳参数的数据结构，从 **sp** 指向的位置开始。第一个参数（C 源代码中最左边的）位于最低地址。每个参数占据至少一个字（32 位）的空间；象 `double` 浮点和 `long long` 的 64 位值必须对齐到八字节边界（包含 64 位标量域的数据结构也是一样）。参数的结构看上去真象一个 C 的 `struct`，但是还要遵守更多的规则。

首先为任何调用分配一个至少 16 字节的参数空间，即使参数没有那么多[8]。

在没有函数原型时，C 语言规定任何宽度小于 `int` 的简单整型参数（即 `char` 和 `short`）要“提升”为一个 `int` 并作为 32 位数据传递。不过这种处理方式只适用于简单类型的参数：并不适用于 `struct` 内部成员。

11.2.3 用寄存器传递参数

分配到参数结构的前 16 个字节（4 个字）单元的参数在寄存器 **a0–a3** (**$4–$7**) 中传递。调用程序通常把该结构的前 16 字节保留为未定义。栈中占据的结构仍然必须保留；被调用函数如果需要的话可以把 **a0–a3** 的值保存到内存中去，而且这样做的时候可以是完全盲目的，无需知道有几个参数，以及什么类型。

[7] SGI 的 n32 和 n64 标准要求堆栈对齐到 16 字节的边界上维护。
[8] 为什么？参见第 11.2.3 节。

在整数寄存器中传递浮点值效率不高，所以取而代之的是，有一个特殊的测试用来识别可以在浮点寄存器中传递的函数参数。

确定什么时候使用以及怎样使用浮点寄存器的标准看上去很奇怪。老式的 C 没有内在机制来检查调用者和被调用者对于函数每个参数的类型是否一致。为了帮助程序员处理这种情况，调用者将参数转换成固定的类型，整数一律转换成 `int`，浮点一律为 `double`。要是程序员连浮点和整数参数都搞混了，那还是没救，但是至少防止了部分混乱。

现代的 C 语言在编译函数调用的时候采用函数原型，由函数原型来确定所有的参数类型。但是即使有了函数原型，也还有一些子程序——最为熟悉的 `printf` ——其参数类型在编译时是未知的，`printf` 在运行时动态发现参数的个数和类型。

MIPS 制定了以下的规则。除非第一个参数是浮点类型，否则不能在 FP 寄存器中传递参数。这样可以保证 `printf()` 等传统函数仍能正常工作：`printf()` 第一个参数是指针所以后续参数全部都分配到整数寄存器中，而且不管参数什么类型 `printf()` 总能够找到所有的参数数据。这个规则也不会使常用的数学函数效率下降，因为这些函数大多数只用浮点参数。

当第一个参数是浮点类型时，该参数就在浮点寄存器中传递。在这种情况下，能够在参数结构前 16 个字节放得下的其它后续浮点参数类型也在浮点寄存器中传递。两个 `double` 占据 16 字节，因此只定义了两个浮点寄存器 **fa0** 和 **fa1** 即 **$f12** 和 **$f14** 用于参数传递。显然没有人认为值得为带有大量单精度参数的函数再定义一个新的规则。

由于 o32 的年龄而导致的一个最大的缺陷是它的寄存器用法和最早期的 MIPS 浮点单元保持兼容，因而只用偶数编号的寄存器来容纳浮点值。双精度值悄悄地延伸到相邻的奇数号寄存器；奇数编号的寄存器只在从存储器或者整数寄存器读写浮点值的时候才用到。o32 导致的寄存器约定并不阻止软件使用后来 CPU 的全部 32 个浮点寄存器，但是那样并不能提高效率。

另外一个比较特殊的地方是，如果定义了一个返回值为结构体类型的函数，并且返回结果在正常使用的两个寄存器里放不下，那么返回值约定要求产生一个指针作为隐含的第一个参数，放在第一个（可见的）参数前之前（参见第 11.2.7 节）。

如果需要写一个调用约定并非简单和显而易见的汇编子程序，可能值得先用 C 编写一个空函数，用“-S”选项编译产生一个汇编文件作为模板。

11.2.4 C 语言库函数中的例子

下面是一个代码的例子：

```
thesame = strncmp("bear", "bearer", 4);
```

我们分别画出参数结构和寄存器（参见图 11.4），当然这个例子中没有参数数据进入存储器，后面我们将看到这样的例子[9]。

因为参数少于 16 字节，所以都能放进寄存器中。

看上去好像用了一个极为复杂的方式将三个参数放进常规寄存器。让我们再来试一下数学库中更容易出错的例子；

```
double  ldexp(double, int);

y = ldexp(x, 23);  /* y = x * (2 ** 23) */
```

图 11.5 给出了相应的参数结构和寄存器值。

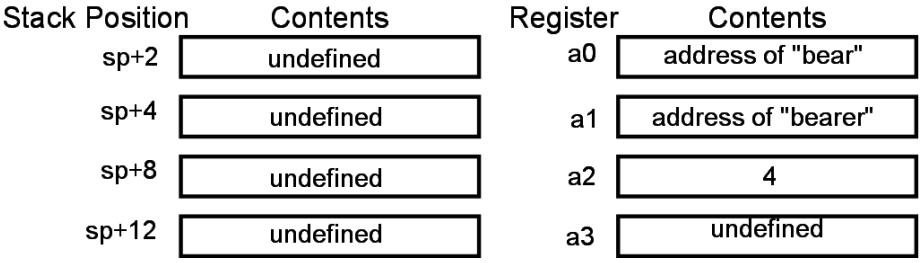

图 11.4: strncmp() 的参数传递，三个非浮点参数

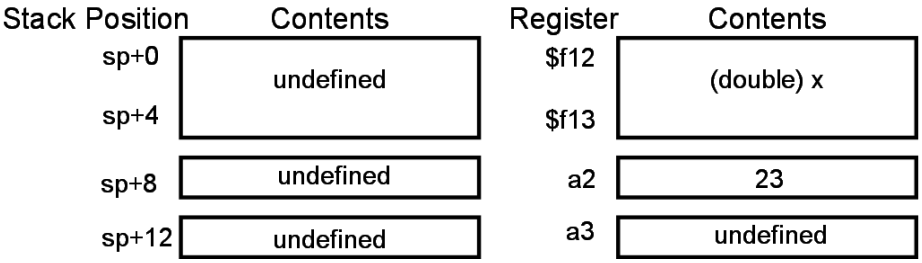

图 11.5: ldexp() 的参数传递：浮点参数

11.2.5 一个反常的例子：传递结构体

C 语言允许用结构体类型作为参数（更常见的是传递指向结构体的指针，但是 C 语言二者都支持）。为了和 MIPS 规则保持一致，被传递的结构体就成为参数结构的一部分，其内部布局和其通常的存储器映像完全相同。在 C 的结构体里，字节和半字紧缩到单个字的存储器单元中，所以当我们通过寄存器来传递概

[9]经过思想斗争，我最后决定在这些图中最好还是把参数从上到下排序。因为栈向下增长，就是说存储器地址沿着页面向下增加，这和我在本书中别的地方画存储器结构的方式正好相反。

念上属于驻留堆栈的结构时，我们不得不用数据装满寄存器以模仿存储器中的数据排布。

这样，如果我们有：

```
struct thing {
    char letter;
    short count;
    int value;
} = {'z', 46, 100000};

(void) processthing(thing);
```

那么就会生成图 11.6 中的参数。

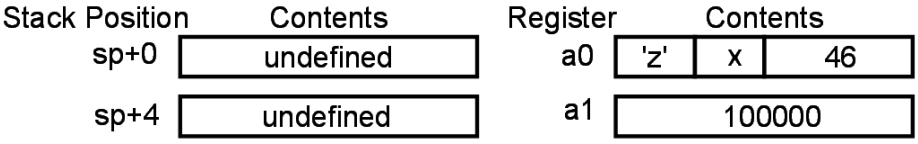

图 11.6: 传递结构类型时的参数

MIPS C 结构的布局让各个域在存储器中的顺序和定义的顺序保持一致（当然必要时为符合对齐规则而进行填充），所以各个域在寄存器中的放置遵循 load/store 指令给出的字节序，随着 CPU 的尾端而不同。图 11.6 中的布局反映了一个大尾端的 CPU，此时结构体中的 `char` 值应当位于参数寄存器的高 8 位，但是和 `short` 压缩到了一个寄存器中。

如果真要传递结构体类型的参数，而且参数中包含短于字的数据类型，那么就应该测试一下这种情况，看看编译器的处理是否正确。

11.2.6 传递不定数量的参数

参数的个数和类型只有在运行时才能确定的函数把约定强调到了极限。考虑下面的例子：

```
printf("length = %f, width = %f, num = %d\n", 1.414, 1.0, 1.2);
```

按照上面的规则，我们可以看出，参数结构和寄存器的内容如图 11.7 所示。

有两点要注意：首先，在 **sp+4** 处的填充对保证 `double` 值的正确对齐是必须的（C 语言规定浮点参数永远按照双精度传递，除非你明确使用类型强制转换或者函数原型来要求用单精度）。注意填充到八字节边界可能导致标准参数寄存器之一被跳过。

其次，因为第一个参数不是浮点值，按照规则不要使用任何浮点寄存器来传递参数。所以第二个参数（与在存储器中同样编码）加载到了两个寄存器 **a2** 和 **a3** 中。

这比表面看上去的用处要大得多!

`printf` 函数的定义用到了 `stdarg.h` 宏包，该宏包为在访问事先不可预知参数的个数和类型时涉及到的寄存器和栈操作提供了一个可移植的封装形式。`printf()` 函数通过获取第一个或者第二个参数的地址解析参数，并在参数结构中顺着存储器地址向上搜索来找到其余参数。

要做到这一点，我们需要说服处理 `printf()` 函数的 C 编译器把寄存器 **a0** 到 **a3** 存放到参数结构中的较浅的位置。有些编译器看到你取参数的地址就会得到提示；ANSI C 编译器应当在函数定义中处理“...”；其它编译器可能需要用些令人讨厌的“pragma”，但是这些都被宏包隐藏了。

现在你可以看出为什么把 `double` 值放进整数寄存器是必要的；这样 `stdarg` 和编译器可以只把 **a0–a3** 存放到参数结构的前 16 个字节中而不用管参数的类型和个数。

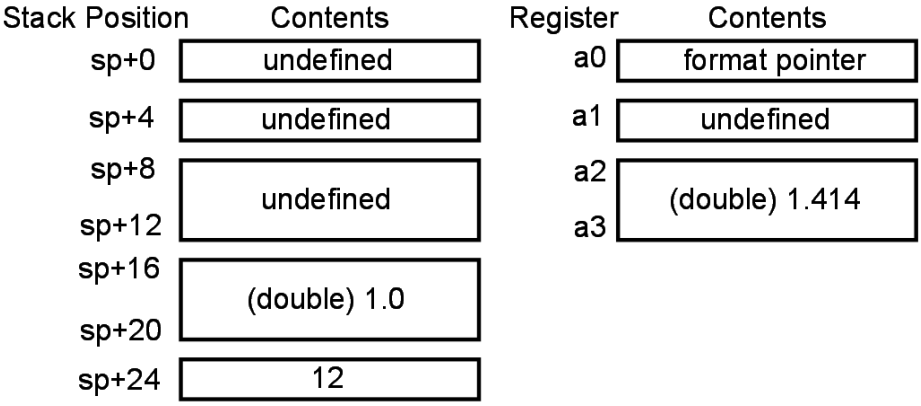

图 11.7: `printf()` 的参数传递

11.2.7 从函数返回一个值

整型或者指针类型的返回值存放在寄存器 **v0**(**$2**) 中。按照 MIPS 约定，寄存器 **v1** (**$3**) 要是保留的，尽管很多编译器根本不用。但是可以期望用在返回 `long long`（64 位整数）类型值的 32 位代码中。对于一个占用五到八个字节空间的结构体值是否用两个寄存器来返回，还有不少争议：GNU C 编译器总是用指针返回超过寄存器大小的结构体，但是规范对此没有明确规定。

任何浮点结果都在寄存器 **$f0** 中返回（32 位 CPU 中如果是双精度值则隐含用到了 **$f1**）。

如果 C 语言中声明的函数返回一个大到返回寄存器 **v1** 和 **v2** 无法容纳的结构体值，就需要采用别的方法。在这种情况下，调用者在自己的栈上为一个匿名的结构体变量分配空间，并在所有显式参数前面加上一个指向该结构体的指针；被调用函数将返回值复制到这个地方。按照一般的参数规则，隐含的第一个参数在函数调用时置入寄存器 **a0**。返回时，**v0** 也指向被返回的结构体。

11.2.8 寄存器用法标准的演化：SGI 的 n32 和 n64

在本节（调用约定和整数寄存器用法），对 n32 和 n64 不加区分[10]。

尽管作了很多努力来保持寄存器约定相似，o32 和 n32/n64 高度不兼容，用不同方式编译的函数无法成功的链接到一起。以下几点概括了 n32/n64 的规则：

- 寄存器中可以传递多达八个参数。
- 参数槽和用于参数传递的寄存器都是 64 位。较短的整数类型都提升为 64 位，就象加载到寄存器一样。
- 调用者不需要为在寄存器中传递的参数分配栈空间。
- 本身就占据前八个参数槽之一的浮点值都通过 FP 寄存器传递，甚至包括数组和结构体中对齐的 `double` 域，只要该域不在 `union` 类型中或者不是 `printf()` 之类参数不确定的函数的可变参数。
- n32 和 n64 认识 16 字节大小的基本对象（比如 `long double` 浮点类型），这样的对象对齐到 16 字节边界。这也要求对于每个函数，栈帧必须重新对齐到 16 个字节的整数倍。

当问题变得复杂（即传递结构体或者数组类型的参数）的时候，寄存器的使用仍然要根据一个假想的参数结构确定，即使此时并没有为前八个参数槽预留栈空间。

n32/n64 约定废除了 o32 的一个约定，就是 o32 关于 `printf()` 等函数要求第一个参数不是浮点的约定以便区别于普通浮点参数的情形。新的约定要求调用者和被调用者代码在编译时要明确知道参数的个数和类型，因此需要有函数原型。

对于 `printf()` 之类的函数，参数的类型在编译时未知，所有的未知参数实际上都是在整数寄存器中传递的。

n32/n64 组织有一套不同的寄存器使用约定；表 11.2 将整数寄存器的使用和 o32 系统作了比较。唯一的实质性差别在于：以前单纯用于临时存储的四个寄存器现在用于传递第五至第八个参数。我对于在临时寄存器间随意而且明显不必要的重新分配名字的做法感到疑惑不解，但是他们就是那样做的。

[10]在 n64 约定下，long 和指针类型都编译为 64 位；而 n32 仅有 long long 类型编译为 64 位。

表 11.2: 新的 SGI 工具中整数寄存器用法的演化

寄存器编号	名字	用法
$0	zero	永远为零
$1	at	汇编临时使用
$2，$3	v0, v1	函数返回值
$4–$7	a0–a3	参数
	o32	n32/n64
	名字 用法	名字 用法
$8–$11	t0–t3 临时存储	a4–a7 参数
$12–$15	t4–t7	t0–t3 参数
$24, $25	t8, t9	t8, t9
$16–$23	s0–s7	保存寄存器
$26, $27	k0, k1	中断/自陷处理程序保留
$28	gp	全局指针
$29	sp	堆栈指针
$30	s8/fp	需要时用作帧指针（不需要时作为另外的保存寄存器）
$31	ra	子程序的返回地址

你或许认为编译后的代码会因为损失了四个原本用于临时存储的寄存器而影响效率，但这只是表象。大多数时候所有参数寄存器以及 **v0** 和 **v1** 寄存器都可以让编译器用作临时存储。另外，到 n32/n64 的这一变化并没有影响哪些寄存器被指定为“保存”（即那些在子程序调用过程中其值得到保存的寄存器）[11]。

浮点寄存器约定（如表 11.3 所示）的变化幅度更大；这并不意外，因为 n32/n64 约定是用于后来的 MIPS CPU 的，这些 CPU 拥有完整的 64 位浮点单元和 32 个完全可用的独立的寄存器[12]。本来 SGI 可以交替使用寄存器以维持某种程度上的兼容性，该公司最后还是决定抛弃现有的大多数规则重新开始。

除了可以用寄存器传递更多的参数之外，n32/n64 标准并没有制定任何依赖于第一个参数是否浮点类型的规则。取而代之的是，根据在参数列表中的位置将参数分配到各个寄存器。让我们重复用一下上面的例子：

```
double  ldexp(double, int);

y = ldexp(x, 23);  /* y = x * (2 ** 23)  */
```

[11] 并不完全如此。在位置无关代码中，函数通过操作 **gp** 寄存器来跟踪一个数据/函数地址表（详情见第 16 章）。在 o32 中，每个函数可以根据自己需要处理 **gp**，就是说在每个函数调用之后你可能需要恢复该寄存器的值。在 n32/n64 中，**gp** 寄存器现在定义为“保存的”。

[12] 所有 MIPS CPU 都有一个模式开关可以让浮点行为完全兼容于老式的 32 位 CPU；n32/n64 假定 CPU 运行时关闭该模式。

表 11.3: o32 及 n32/n64 约定中的浮点寄存器用法

寄存器号	o32 用法	
$f0, $f2	返回值；fv1 仅用于 FORTRAN 中的复数数据类型，C 语言没有。	
$f4, $f6,$f8,$f10	临时存储——函数可以不必保存就直接使用	
$f12,$f14	参数	
$f16,$f18	临时存储	
$f20,$f22,$f24, $f26,$f28,$f30	保存寄存器——函数必须保存和恢复要写入的寄存器，使得它们适用于存续时间跨越函数调用的长期的值	
寄存器号	**n32 用法**	**n64 用法**
$f0,$f2	返回值——$f2 仅用于返回恰好包含两个浮点值的结构体；这是处理 FORTRAN 复数的一个特例	
$f1,$f3 $f4–$f10	临时存储	
$f12–$f19	参数	
$f20–$f23	偶数号（$f20–$f30）为临时存储；奇数号（$f21–$f31）要保存	临时存储
$f24–$f31		保存寄存器

在 n64 中，双精度的参数会放进浮点寄存器 **$f12** 而整数值 23（“符号扩展”至 64 位）放进 **$a1**，即用来存放第二个槽中参数的整数寄存器。对这些参数不保留栈空间。

虽然 n32/n64 可以处理浮点和其它值的任意混合，而且仍然将前八个参数中的 `double` 类型放入浮点寄存器，还是有一些规则要小心注意。被 `union` 碰过的参数（因而不一定真的是 `double` 类型）被排除在外，还有参数个数不定的函数的参数也被排除在外。因为函数本身和调用程序必须做同样的决定（否则无法处理），这就依赖于拥有正确的函数原型。如今，这在大多数情况下是一个合理的假设。

11.2.9 栈的布局、栈帧以及对调试器的支持

图 11.8 给出了一个 MIPS 函数的栈帧示意图（我们重新回到让栈向下增长，顶部表示内存高地址的画法）。你应当能认出传统 MIPS 函数调用规定为该函数参数所保留的前四个字的空间——新的调用约定只提供实际需要的空间。

图中的灰色阴影区域表示该函数自身所用的空间；粗线上方的白色区域属于调用函数。栈帧中的全部灰色区域都是可选的，有些函数根本不需要；这种简单的函数并不需要用到栈。我们在本章剩余的部分的例子中会碰到这样一些函数。

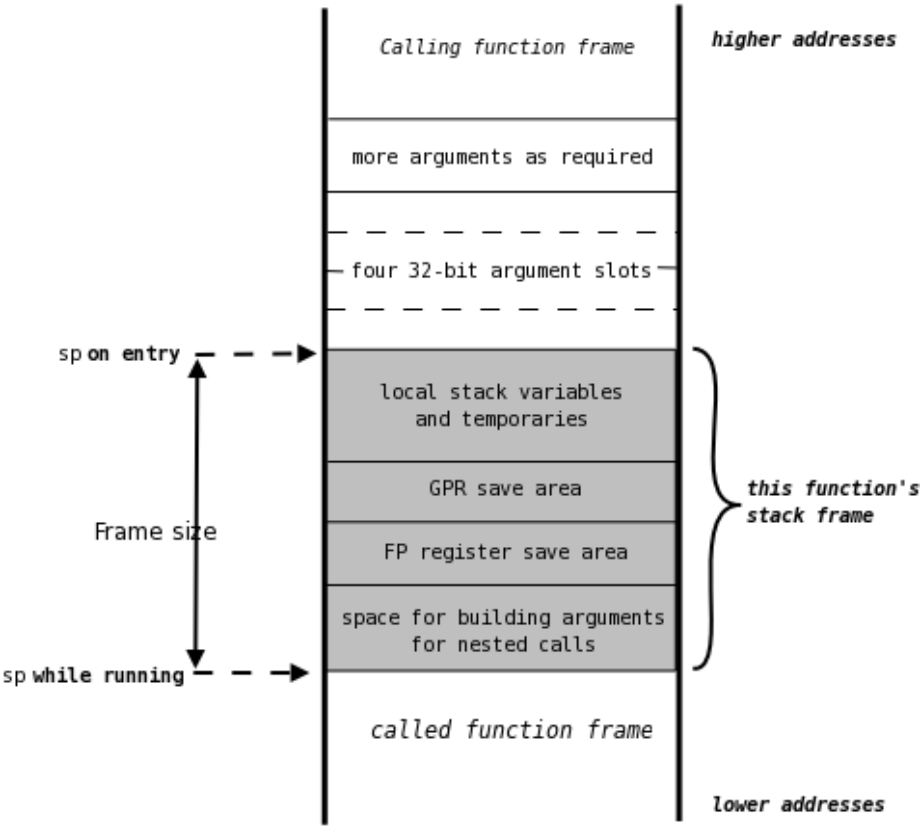

图 11.8: 非叶子函数的栈帧

除了参数（其布局必须与调用函数一致）之外，还有栈的结构也属于函数私有。我们需要一个标准的栈数据安排的唯一原因是为了便于调试和诊断工具在栈中导航。如果我们中断一个正在运行的程序进行调试，我们很想要能够沿着栈向上返回，显示出到我们的断点一路上的函数调用，以及传递给这些函数的参数。除此而外，我们还想要能够在调试器环境中沿着栈向上往回走几步，找出变量在新的环境中的值——即使该段代码被优化的编译器编译成在寄存器中维护变量值，这样做依然有效。

要进行这种分析，调试器必须知道一个标准的栈布局并得到相关信息能够看到每个栈帧成员的大小以及这些成员的内部布局。如果一个函数在栈上某处因为要用 **s0** 而保存了 **s0** 的值，调试器需要知道从哪里找到保存的值。

在 CISC 体系结构中，常常有一条复杂的函数调用指令来维护类似于图 11.8 的栈帧，还要额外加上对应我们图中标记为“sp on entry” 的帧指针寄存器。在这种 CPU 中，调用者的帧指针将会存储在栈中某个已知位置，允许调试器通过分析一个简单的链表而跳过栈中几个位置。但是在 MIPS CPU 中这些额外的运行时工作都没有；大多数时候，编译器知道在函数开头对栈指针减去多少，在函

数返回之前增加多少。

那么在最小的 MIPS 栈帧中，调试器从什么地方找出数据存放在哪里呢？有些调试器极为神奇，甚至能够分析一个函数的头几条指令算出栈帧的大小并定位保存的返回地址。但是大多数工具链在目标代码中至少传递部分栈帧的信息，由适当的汇编伪指令写入目标代码。

因为混合了伪指令后高度依赖于工具链，值得定义一个函数首部和尾部宏，这样可以不必记住细节，而且必要时容易换到别的工具链。大多数工具链提供了现成的宏可以直接使用；你可以看到下面的例子中用到的名为 LEAF 和 NESTED 的简单宏。

我们并不想提供用于调试的汇编伪指令背后的细节：抱歉。你最好的做法可能是编译一些代码，然后研究一下编译器的汇编输出。

我们把函数分成三种类型并给出三种不同方法，应该涵盖了你需要的一切。

叶子函数

如果一个函数不包括对别的函数的调用，就叫做*叶子函数*。叶子函数不必担心设置参数结构，可以安全的在非保留的寄存器 **t0–t7**、**a0–a3** 以及 **v0** 和 **v1** 中维护数据。如果觉得需要也可以用栈保存数据，但是能够也应当把返回地址留在寄存器 **ra** 中并直接返回到 **ra** [13]。

出于性能优化或者访问 C 语言没有的特性而用汇编写的大多数函数都会是叶子函数；许多根本就不用栈空间。这种函数的声明非常简单，例如：

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF(myleaf)
    ...
    <your code goes here>
    ...
    j   ra
END(myleaf)
```

多数工具链可以把你的汇编源代码在汇编之前传递给 C 的宏处理器——Unix 之类的工具根据文件的扩展名来决定。文件 `mips/asm.h` 和 `mips/regdef.h` 包含有用的宏（比如上面的 LEAF 和 END）来声明全局函数和数据；同时也允许使用软件寄存器名，例如用 **a0** 而不必用 **$4**。

LEAF 和 END 宏将辅助调试器周遍历你的函数而需要告知汇编器的全部信息打包到了一起：它们并不提供除了函数名字外的其它任何信息。

[13] 返回地址存放于别处也完全能够工作，但那样调试器就找不着了。

非叶子函数

非叶子函数就是包含了对别的函数的调用的函数。正常情况下，函数一开始的代码（函数首部）把 **sp** 设置到函数内部使用的低水线；低水线就是嵌套的函数调用的参数结构的基址。我们也需要栈空间来保存该函数用到的任何"saved"寄存器 **s0–s8** 新进来的值。也要为 **ra** 和自动变量（即基于栈的局部变量），还有该函数在自己的调用过程中需要保存的其它寄存器的值预留栈空间。（如果参数寄存器 **a0–a3** 的值需要保存，可以保存到参数结构的标准位置上。）

注意到因为 **sp** 只设置一次（在函数首部代码中），栈中保留的所有位置都可以用从 **sp** 开始的固定偏移量引用。

为了说明这一点，我们将定义一个极为简单的 C 模块。函数 `nonleaf()` 带有五个参数（这样从栈上至少要传递一个参数），它要调用另一个带五个参数的函数，其中一个参数是指向栈中地址的一个指针。这些应当给让你对 ABI 是怎样工作的有足够多的感性认识。

```
extern nested(int a, int b, int c, int d, int *e);
extern nonleaf(int a, int b, int c, int d, int e)
{
    nested(d, b, c, a, &e);
}
```

这给出一个类似表 11.4 所示的栈结构。

表 11.4: nonleaf() 的栈布局

	48	e
	44	(reserved for c/a3)
	40	(reserved for b/a2)
	36	(reserved for a/a1)
sp on entry =>	32	(reserved for a/a0)
	28	(pad to 8 bytes)
	24	saved ra
	20	(pad to 8 bytes)
	16	&e
	12	(reserved for a/a3)
	8	(reserved for c/a2)
	4	(reserved for b/a1)
sp running =>	0	(reserved for d/a0)

你可以看到那些因为相应的参数在寄存器中而保留但未用的参数槽的位置。你也可以看到，因为最低的保存的寄存器和运行的栈帧的底部都必须对齐到八字节边界而引入填充的地方。

用 GNU C 编译器把 nonleaf.c 编译成汇编代码就得到：

```
    .file 1  "nonleaf.c"
    .section .mdebug .abi32
    .previous
    .text
    .align  2
    .globl  nonleaf
    .set    nomips16
    .ent    nonleaf
nonleaf:
    .frame  $sp,32,$31        # vars= 0, regs= 1/0, args= 24, gp= 0
    .mask   0x80000000, -8
    .fmask  0x00000000, 0
    .set    noreorder
    .set    nomacro

    addiu   $sp, $sp, -32
    sw      $31, 24($sp)
    move    $2, $4
    addiu   $3, $sp, 48
    sw      $3, 16($sp)
    move    $4, $7
    jal     nested
    move    $7, $2

    lw      $31, 24($sp)
    j       $31
    addiu   $sp, $sp, 32

    .set    macro
    .set    reorder
    .end    noleaf
    .ident      "GCC:  (GNU) 3.4.4 mipssde-6.03.01-20051114"
```

我们可能需要分成几部分过一遍。

```
    .file 1  "nonleaf.c"
    .section .mdebug .abi32
    .previous
```

文件的开始是一些用于调试的基本信息（如果我们编译时加上“-g”选项，就会有比这多很多的调试信息）。我们不做进一步解释，抱歉!

```
    .text
    .align  2
    .globl  nonleaf
    .set    nomips16
    .ent    nonleaf
nonleaf:
```

目标代码和链接用的信息。我们生成的是代码，所以放进目标文件的 **.text** 区，函数入口名字是一个全局的符号同时也是一个入口点。也有入口点的标号。

```
    .frame  $sp,32,$31        # vars= 0, regs= 1/0, args= 24, gp= 0
    .mask   0x80000000, -8
    .fmask  0x00000000, 0
```

有关栈的信息。我们为这个函数生成 32 个字节的栈帧，如表 11.4 所示。唯一要保存的寄存器就是 **$31** 即返回地址。这也在 **.mask** 中保存寄存器的位图的位 31 中看得出来，-8 代表用于保存通用寄存器的堆栈块的位置。**.fmask** 对浮点寄存器完成同样的工作：这里没有保存任何浮点寄存器，所以就是零。

```
    .set    noreorder
    .set    nomacro
```

这两行告诉汇编器在这里由编译器负责。汇编器不要移动指令来填充分支延迟槽，或者解释“宏指令”（机器码中展开为多条指令的汇编指令）。

现在该看到一些代码了：

```
    addiu   $sp, $sp, -32
    sw      $31, 24($sp)
```

调整栈指针并且保存从 `nonleaf()` 调用的返回地址。

```
    move    $2, $4
    addiu   $3, $sp, 48
    sw      $3, 16($sp)
    move    $4, $7
    jal     nested
    move    $7, $2
```

调整参数然后调用 `nested()`。

```
    lw      $31, 24($sp)
    j       $31
    addiu   $sp, $sp, 32
```

检索返回地址并且返回。在延迟槽中恢复栈指针。

```
.set    macro
.set    reorder
.end    noleaf
.ident      "GCC:  (GNU) 3.4.4 mipssde-6.03.01-20051114"
```

重新打开刚才关闭的汇编特性，并且包含一个标识编译器版本的字符串。

复杂的栈需要的帧指针

在上面描述的栈帧中，编译器只用一个保留寄存器 **sp** 就能管理整个栈。熟悉别的体系结构的读者知道维护堆栈常常需要用两个寄存器：栈指针 **sp** 标识栈的底部，帧指针指向有函数首部创建的数据结构。然而，只要编译器能够在函数首部代码中分配函数需要的全部栈空间，就能在首部代码中降低 **sp** 让它在函数整个执行过程中指向栈中一个固定的偏移位置。如果这样，局部栈帧上的一切和 **sp** 之间的偏移量都是在编译时已知的，就不需要帧指针了。但是有时候你需要在运行时操作栈指针：图 11.9 显示 MIPS 怎样分配一个帧指针来处理这种需要。

什么会导致一个无法预测的栈指针呢？在某些语言中，甚至 C 语言的某些扩展中，可以创建其大小在运行时变化的动态变量。许多 C 编译器可以通过有用的内建函数 `alloca()` 即时分配栈空间[14]。在这种情况下，函数首部代码占用另一个寄存器 **s8**（常用的别名就是 **fp**），把它设置为 **sp** 刚开始的值。

因为 **fp**（也就是 **s8**）属于保留寄存器之一，函数首部必须保存其原来的值，就好像我们把 **s8** 用作了一个子程序变量。采用帧指针编译的函数中，所有局部栈帧的引用都通过 **fp**，这样若编译器需要降低 **sp** 来给运行时计算出大小的变量分配空间，就可以直接做了。

注意如果函数有一个嵌套的调用用了许多参数以至于需要在栈上传递数据，那就要用到与 **sp** 的关系。

这个窍门的一个巧妙的特点就是不论是帧指针函数的调用者，还是被它调用的函数，都不把它当作一个特殊的寄存器处理。调用的函数本来就要保存 **fp**，因为它是一个由被调用函数保存的寄存器；被调用函数可见的栈帧部分看上去本该如此。

汇编爱好者可能乐于看出，当用 `alloca()` 创建空间时，返回的地址实际上比 **sp** 高一点，因为编译器仍然为函数调用要求的最大的参数结构保留了空间。

当局部变量的空间增长大到栈帧上的有些数据离 **sp** 太远，以至于无法用一条单个的 MIPS load/store 指令访问的时候（偏移量限制为 ±32-KB），也有些工具在这个时候采用基于 **fp** 的栈帧。

[14] 实际上，`alloca()` 的有些实现并不在局部栈上分配空间，有些则是纯粹的库函数（这意味着你不需要因为可移植性的原因而回避 `alloca()`。）但是用栈空间实现 `alloca()` 的编译器生成的代码运行快一些。

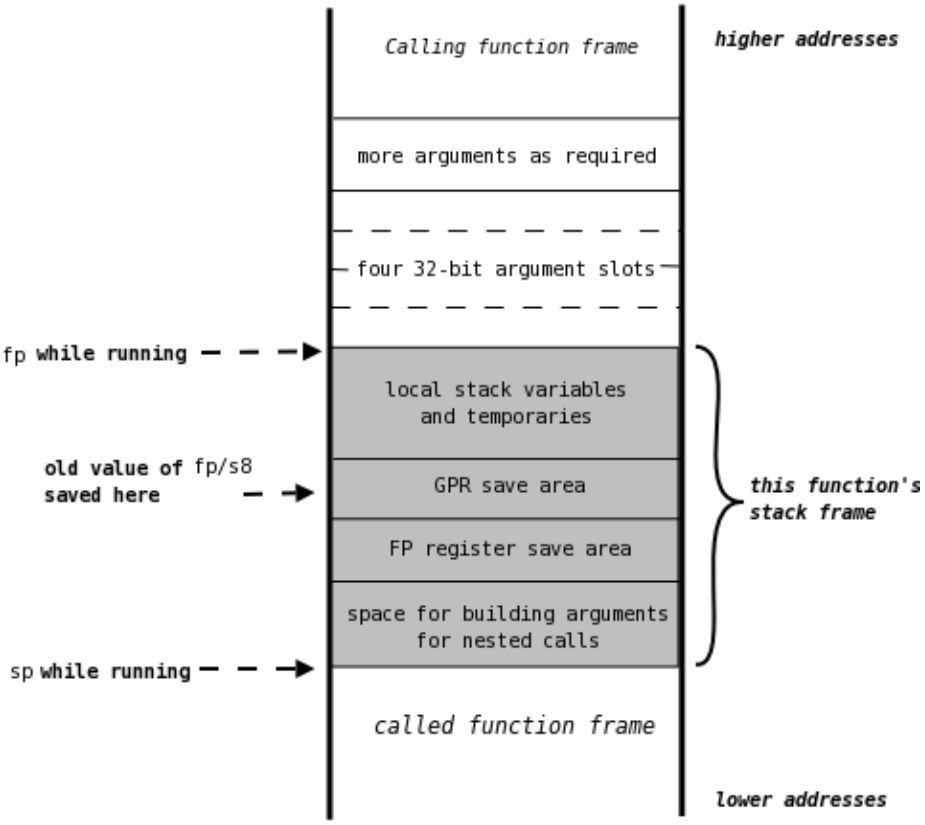

图 11.9: 使用独立帧指针寄存器的栈帧

11.2.10　个数不固定的参数和 stdargs

如果你要定义一个参数个数不确定的新函数，请用工具包的 stdarg.h 宏包（ANSI 标准规定必须有）。该宏包提供了宏——可能还有函数——va_start()、va_end()、va_arg()。要了解怎么用，看一下实现 printf() 的宏包就知道了：

```
int printf(const char *format, ...)
{
    va_list arg;
    int n;

    va_start(arg, format);
    n = vfprintf(stdout, format, arg);
    va_end(arg);

    return n;
}
```

一旦我们调用 `va_start()`，我们就可以提取出我们想要的参数。所以在为 `printf()` 实现格式转换的代码中间某个地方，你会看到下面的代码被用来解析下一个参数，这里假定该参数是双精度浮点类型：

```
...
d = va_arg(ap, double);
...
```

决不要去用汇编语言写参数个数不确定的函数——不值得为此而在可移植性方面引入许多麻烦。

第 12 章　MIPS 调试——调试和剖析

当你构建一个低成本、低功耗的嵌入式设备或者消费类电子产品的时候，把一切对最后的应用没有贡献的特性都去掉是有意义的。

不是吗？

用市场营销的话来说，首先要考虑从这些设备挣钱的方式：随着设备批量投入市场，价格下降很快（比如象已经发生在 DVD 播放器身上的那样）。产品生命周期变短，随之开发时间也要缩短。

这就有了一个矛盾。一个高度裁减的硬件系统很不利于开发和测试。SoC 芯片的规模经济提供了一个解决办法。大多数 SoC 可以在不增加产品成本的前提下比预定的功能增加百分之几的晶体管，因为晶体管的数目主要受限于电路设计的复杂性和保证芯片一次成功的困难性。如果 SoC 元件供应商能在不将风险转嫁给 SoC 制造商的前提下包含更多的辅助开发逻辑，那么大家都是赢家。

所以现代的 MIPS CPU 一造出来就带有一定的硬件用于辅助开发。不同的制造商用不同的方法来满足竞争需求，这样一来——对于开发人员和开发工具生产商来说不幸的是——这些特性都是可选的，你的系统不一定有。但是最近几年的一个确定的趋势是加入越来越多的辅助开发逻辑。

开发人员的受益来自于可以从中得到他们的系统的三个不同方面的信息，我们可以合理的把这些信息分为“调试(debug)”、“跟踪(trace)”和“剖析(profile)”三方面。

调试——得知系统的确切状态

在纠正与系统工作有关的逻辑错误的时候，要是有什么办法能够一直挖到底，甚至有时能到最底层（一条一条指令）来察看究竟发生了什么事，那就太有用了。

这就是软件调试器的工作。要在复杂的系统中能用，调试器需要能够访问软件的源代码、二进制代码，而且要能把二者对应起来。将调试事件和源代码对应起来是一个大问题，我们这里不去讨论，只是指出这样一个调试器无法全部驻留在待测的嵌入式系统里面，所以调试器需要某种主机-目标机连接。在 CPU 层

次，调试器需要访问程序的内存和 CPU 的状态，以及通过在适当的地方安排到调试器的软件自陷来控制程序执行的能力。

MIPS 规范定义了一个如下所述的 CPU 片内辅助调试单元，名叫 EJTAG。但是 MIPS CPU 也保留了 EJTAG 之前的调试特性，之所以还保留这些是因为它们对某些类型的调试软件用起来更熟悉方便，我们也会提到这些。

跟踪——刚才都发生了什么？

获取一些系统如何从“正确”到走到错误的行为的总体过程，对于了解系统运行到了哪一步是很有用的。整个系统的状态大得难以管理，事前很难知道哪些状态更重要。但对 CPU 而言，察看最近执行过的指令肯定很有意义。

当 CPU 速度快到每秒运行几十亿条指令的时候，要保存足够多的状态信息并涵盖足够长的时间跨度和足够多的细节、并以足够快的速度把信息送出芯片以免停止 CPU 或者丢失部分跟踪信息，是一个巨大的挑战。

MIPS CPU 中使用的系统叫做 PDtrace，和 EJTAG 调试块接在一起。

剖析——一段时间内的系统行为统计信息

如果说提供调试器是因为程序员不能在头脑中记住软件的每一个可能的细节的意义的话，那么剖析器存在的意义就是因为要想象出整个子系统的行为更加困难。模块化的程序设计实践使得正确构建系统成为可能，它将程序员从 CPU 具体怎样执行任务的知识隔离开来——但是对于有一定复杂度的系统，这就意味着程序员对于按下一个按钮是否导致满意的响应还是未知的等待，只有一个极为模糊的概念。

在大尺度上提供系统的行为和时间的统计信息的软件工具就叫做剖析器。剖析器的工作是通过向系统提交程序以及对一段时间内的特定事件进行计数来完成的。

EJTAG 系统包含一个叫做“PC 取样”的可选项，能提供一些有用的信息；但是MIPS CPU 还有几个由目标机上软件控制的性能计数器。这些计数器及其控制都以 CP0 寄存器的方式提供，下面有进一步的描述。

当开发人员运行一个系统进行调试、跟踪或剖析的时候，提供底层信息的硬件本身可能改变系统的行为。这点没有好处：我们希望辅助开发功能尽可能不要影响系统的行为，即为非破坏性的(non-intrusive)。

我们前面说过，SoC 上用于辅助开发的晶体管极端便宜，可以廉价获得。难以用低成本提供的是 SoC 和开发系统之间的连接器，由它将调试信息带回来分析。

在老式的系统中，通常用额外的串口或者别的系统 I/O 来连接目标板和宿主机。但更多的时候涉及到跟目标系统软件之间关于共享连接器和最终系统中起作用的设备之间的复杂协议。

这种做法会影响系统本身的行为：最好是有个单独的调试连接。理想情况下（为了减少对系统设计的冲击，并且建立一个标准以便于构造工具），调试连接应当直接来自 CPU。但是带有 CPU 的 SoC 通常位于系统的核心：晶体管虽说便宜，但是增加引脚的成本却很贵。

不过在这点上我们很幸运；别人的问题给我们提供了一个解决方法。电子系统通常是焊接在一起的，而焊接过程的缺陷率很高。大多数电子线路板在用于产品之前要进行自动测试。复杂的设备上都预留了几个引脚用于测试。广泛选用的测试连接标准就是 JTAG，这样 MIPS 选择复用 JTAG 引脚来连接开发子系统，反正它们在运行时也没有用。这就是调试单元叫做 EJTAG 的原因。

但是调试硬件仍然可以用于传统的调试器——一种运行在目标系统上采用某种传统设备进行通信的调试器。（例如）深陷于高级操作系统的调试器很可能大部分运行于目标设备上。参见第 12.1.13 节有关的注解，以了解怎样让运行于目标设备上的调试器使用调试单元。

12.1 "EJTAG"片上调试单元

EJTAG 单元是 CPU 内用于构建调试和跟踪工具的资源的一个松散集成的集合。如我们上面所说，EJTAG 通过复用每个 SoC 内已经提供的用于芯片测试的 JTAG 引脚[1]找到了一个不影响被测设备的跟主机的连接方法（并由此而得名）。

这样一个调试单元要求有：

- 和主机的物理连接——一般是通过用一些通用的网络或者连线接到调试主机的某种探接设备。该设备接到 SoC 的 JTAG 引脚上，通常称其为接探器*(probe)*，或简称探针。

- 主机/探针"远程控制" CPU 的能力。那是通过让 CPU 从一个 *dmseg* 的内存区域执行代码实现的，在该区域 CPU 的读写都是由探针控制的。dmseg 是特殊的内存窗口 *dseg* 的一部分，在调试模式中打开，如第 12.1.6 节所述。

  虽然探针在给最小系统提供调试资源方面最有效，EJTAG 资源也完全可以由运行在本地 CPU 上的纯软件的调试器使用。我们后面回头再讲这个问题。

- 调试专用的异常。在 MIPS EJTAG 中，这是一个由特殊的调试模式标志的特殊的超级异常，你可以用 EJTAG 调试器来调试整个系统，甚至它自己的普通异常处理程序。参见第 12.1.4 节。

[1] 当然如果你付得起，提供一个独立于硬件测试引脚的 EJTAG 连接会极为有用。

- 一定数量的 *EJTAG* 断点，可以独立编程来匹配地址乃至数据的硬件监控器[2]。当 CPU 取指或者读写数据时，取指或者读写的地址和数据与活动的断点相比较，若匹配则导致一个调试异常。

  断点控制寄存器可以有许多个，以至于难以加进 CP0 寄存器集，所以通过存储器映射到 dmseg（用于 EJTAG 寄存器的子区域叫做 *drseg*）。

- 你也可以从一条调试断点指令 **sdbbp**、一个外部信号 *DINT* 或者在 EJTAG 探针翻转了某个已知的控制位[3]的时候得到一个调试异常。

- 用于剖析而不是调试的时候，你可以用该工具指导 EJTAG 单元周期性地记录当前执行指令的地址（*PC* 采样），并从 EJTAG 探针获得这些取样信号，参见第 12.1.12 节。

在这些基础之上可以构建强大的调试能力。

12.1.1 EJTAG 的历史

回到微处理器的石器时代（1980 年之前），那时可以从一个在线仿真器（ICE——in-circuit emulator）获得细粒度的调试。ICE 就是一个用来取代 CPU 的插件，带有巨大的专用线缆，看上去象一堆充满电子元件的架板。ICE 可以象 CPU 一样执行指令，不同之处是可以停下来查看。

ICE 方法当时能够有效是因为微处理器非常昂贵而且易坏，所以常常通过一个芯片管座插接而不是直接焊上去——就如同今天的 PC 微处理器这样因为很昂贵以至于不会焊到主板上。

要让复杂的 ICE 电路运行得和简单的 CPU 一样快总是很难的，随着处理器速度越来越快，情况变得更糟。ICE 盒子一开始就很贵，现在更贵了：一旦价格上到几万美元，许多开发团队就会想方设法不用 ICE 进行调试。进入 1980 年代以后，这种情况逐渐毁掉了 ICE 的市场。

所以 ICE 制造商需要类似于 EJTAG 的东西。加上一个片上调试单元后，把关键路径（可能限制了 CPU 的速度）限于局部范围，使得系统可以全速运行。ICE 制造商懂得怎样做连接所需要的探针和运行用的复杂的主机软件。这点很好，但是他们碰到了一个问题：他们要么不得不把这个微小的探针及软件卖到 10 000 美元，要么改变盈利模式薄利多销。技术的变革不难，难的是习惯上的改变。ICE 制造商试图维持高价，就被市场绕开了。

是 LSI 公司第一个在他们的片上系统的 MIPS CPU 中开创性地加进了片上调试单元。在一些 ICE 制造商的帮助下，他们开发了 EJTAG 的规范，基本跟今天的差不多。但是如今探针的成本不到 1000 美元，而且可以接到你选择的任何软件调试器上工作。

[2]软件工程师习惯上用"断点"一词来表示支持调试的指令，通常把硬件监控器叫做"观察点，"这样更有助于区分两个概念。但是所有的 EJTAG 文档都称为断点，所以我们也这么叫。

[3]或者如某个同事痛心的说，正在调试系统的时候不小心碰了下探针

12.1.2 怎样控制 CPU

当 CPU 从 dmseg 区域读取指令的时候，探针就获得控制权。这听起来好象你不得不等待 CPU 的软件送上门来跑到该区域来执行才可以，但其实不然。

- 为了响应调试器的启动代码，探针会用其 JTAG 连接设置一个内部的标志，将调试异常入口点移到 `0xFF20 0200` ——恰好位于 dmseg 区域。

  探针可以直接产生一个调试断点或者设置一个硬件观察点等待软件碰上去。

- 探针可以发送一个 **EJTAGBOOT** 命令（下一节再讲命令）然后等到下次 CPU 复位之后，CPU 就从 dmseg 读取指令。其实有一组特性可以让你重新启动 CPU 并获得完全控制，甚至可以通过 JTAG 连接下载新软件。

一旦探针开始向 CPU 发送指令，就可以让 CPU 做任何事情。特别是，可以发送一组指令序列来（从 CPU 寄存器或者内存）读出数据并且经过 dmseg 空间写回到探针。

12.1.3 通过 EJTAG 的调试通信

你的宿主机通过一个黑盒子——探针连接待测系统。芯片的 JTAG 引脚让探针能够访问 CPU 内部的特殊寄存器。

JTAG 标准是芯片制造商发明用来简化电路板测试的，大多数复杂的芯片已经为 JTAG 引脚安排了空间；让这些引脚同时服务于软件调试要比争取新的引脚容易。在 1970 年代（JTAG 诞生的年代）每增加一个晶体管都将显著增加成本。JTAG 的设计是为了最大限度的减少接收器的逻辑复杂度，接收器基本上就是一个移位寄存器。[4]

有一个控制/数据选通输入线、数据输入线和数据输出线。命令码发送到控制寄存器，大多数命令码仅仅是选择不同的数据寄存器。JTAG 命令码被称为"指令"，但是经常和数据寄存器的名字互换使用——当跟软件工程师讲的时候这种叫法常常引起混淆。

这种非常简单的硬件的净效果就是 JTAG 提供了一种读写 EJTAG 单元内部寄存器的方式。EJTAG 单元提供了若干条指令供探针用于处理 CPU，表 12.1 对此作了一个概括。

12.1.4 调试模式

调试模式是一个特殊的 CPU 状态，很象异常模式（但更甚于异常模式）。CPU 在调试异常发生之后进入调试模式。调试异常可以由一条 **addbp** 指

[4] 对于不了解的读者，移位寄存器就是一个串行存取的一位宽度的存储器。当一位数据随着时钟进来时，其它位都上移一位以腾出地方。在硬件中，移位寄存器常常也是并行接线的（就是说，全部位的状态反映在一组接线上）。

表 12.1: 用于 EJTAG 单元的 JTAG 指令

JTAG"指令"	**描述**
IDCODE	读出 MIPS CPU 和版本——对软件没多少兴趣，此处不进一步讲。
ImpCode	读表示实现了哪些 EJTAG 选项的位域。
EJTAG_ADDRESS EJTAG_DATA	（读/写）合在一起，允许探针为响应取指和在第 12.1.6 节所述的 dmseg 区域的数据读写而提供或者接收数据。
EJTAG_CONTROL	为探针读写的标志和控制位域的集合。
EJTAG_BOOT NORMALBOOT	EJTAGBOOT 指令导致下次 CPU 复位时从探针执行；由被称为 ProbEn、ProbTrap 和 EjtagBrk 的 EJTAG_CONTROL 位控制，但是你要仔细阅读详细的手册找出具体细节。NORMALBOOT 指令恢复正常的 CPU 启动。
FASTDATA	用来加速和探针间的多字数据传输的特殊存取方式。探针读写由 EJTAG_CONTROL(PrAcc) 和 EJTAG_DATA 构成的 33 位寄存器。
TCBCONTROLA TCBCONTROLB TCBADDRESS	访问用来控制"PDTrace"指令跟踪输出的寄存器。参见 12.3 节，但是并没有详细介绍这些可用 JTAG 访问的寄存器。

令、碰到一个 EJTAG 断点寄存器、外部的"调试中断"信号 *DINT* 或者单步运行（后者比较特殊，下面将简要介绍）引发。调试模式状态是可见的，并且可以通过 CP0 寄存器位 **Debug(DM)** 来控制——第 12.1.7 节详细介绍 EJTAG 的 CP0 寄存器。调试模式（象异常模式一样）隐含的禁止所有的常规中断。调试模式中的地址映射也有所变化，使得你可以访问下述的 dseg 区。许多的异常在调试模式中都不会发生；少数发生的异常也会特殊运行——参见下一节。

连接有适当探针的 CPU 可以设置成让调试异常入口点位于 dmseg 区，运行由探针提供给的指令。如果没有连接探针，调试异常入口点就位于 ROM 中（参见表 5.1，其中给出了 MIPS 所有异常的入口点。）

调试模式中的异常

软件调试器很可能编码成避免引发异常的方式（例如在软件中检查地址，而不是冒地址或 TLB 异常的风险）。所以在调试模式中，可以合理地忽略在正常情况下会引起异常的很多条件：中断、调试异常（执行 **sdbbp** 引起的异常除外）和许多其它条件。

但是有几个发生在调试模式中的异常会变成*嵌套的调试异常* ——对使用 EJTAG 探针的调试器可能最有价值的一个工具。

在这样一个嵌套的调试异常中，CPU 跳转到调试异常入口点，仍然保持在调试模式。**Debug(DExcCode)** 域记录了嵌套异常的原因，**DEPC** 记录了异常模式代码重新开始的地址。要是调试器进入调试模式后没有很快保存一份原来的 **DEPC** 的拷贝，这种做法对调试器就无效。但是调试器可能都会那样做！要从一个嵌套的调试异常返回，并不是用 **deret**（它可能会让你不适当地退出调试模式）；而是从 **DEPC** 中获得地址然后用一条 **jr** 指令。

12.1.5 单步运行

当单步运行位 **Debug(SSt)** 置位并且用一条 **deret** 指令让控制从调试模式返回时，由 **DERET** 选择的指令将在非调试环境下执行[5]；然后按顺序取出的下一条指令就会发生一次调试异常。

因为有一条指令运行于正常模式，可能导致非调试异常；在这种情况下，"按顺序下一条指令"就会成为异常处理程序的第一条指令，你会得到一个单步调试异常，其 **DEPC** 指向异常处理程序。

12.1.6 dmseg 存储器译码区域

EJTAG 需要用一些存储器空间，容纳其大量的断点管理寄存器（对 CP0 来说太多了），还要有为探针映射的用于通信的空间。这段存储器空间在 CPU 处于调试模式时虚拟地址映射的顶部附近，如表 12.2 所示。

在表中，你可以看到：

- dseg: 整个调试模式专用的内存区，看上去重叠了 CPU 处于调试模式时上部的内核访问映射区（kseg2）。调试软件可以通过设置 **Debug(LSNM)** 来读取被"埋在下面"的 kseg2 映射的地址——但是调试友好的操作系统应当会避免把这部分内存区用于其它目的。

- dmseg: 读写都是通过探针实现的内存区域。但是如果没有接活动的探针，或者如果 **DCR(PE)** 位清零，那么访问这里的读写就和通常的 kseg2 同样处理。

- drseg: 是访问调试单元的主寄存器组的地方。对 drseg 的访问并不离开 CPU。"drseg"中的寄存器宽度为一个字，只能用字宽度（32 位）加载和存储指令访问。

- fastdata: 是 dmseg 的一个角落，在这里探针映射的读写采用一个效率更高的 JTAG 块模式探测协议，降低了 JTAG 的数据流量，允许更快的数据传输。本书中不讲具体实现的细节。

[5]如果 **DERET** 指向一条分支指令，分支指令及延迟槽中的指令都会正常执行。

表 12.2: EJTAG 调试存储区映射（dseg）

虚拟地址	区域/子区域			位置/寄存器	虚拟地址
0xC000 0000					0xC000 0000
0xFF1F FFFF					0xFF1F FFFF
0xFF20 0000				**fastdata**	0xFF20 0000
0xFF20 000F				**fastdata**	0xFF20 000F
0xFF20 0010					0xFF20 0010
0xFF20 0200			dmseg	debug entry	0xFF20 0200
0xFF2F FFFF					0xFF20 0200
0xFF30 0000				DCR register	0xFF30 0000
0xFF30 1000				IBS register	0xFF30 1000
				I-breakpoint #1 regs	
0xFF30 1100				IBA1	0xFF30 1100
0xFF30 1108				IBM1	0xFF30 1108
0xFF30 1110				IBASID1	0xFF30 1110
0xFF30 1118				IBC1	0xFF30 1118
				I-breakpoint #2 regs	
0xFF30 1200				IBA2	0xFF30 1200
0xFF30 1208				IBM2	0xFF30 1208
0xFF30 1210	kseg2			IBASID2	0xFF30 1210
0xFF30 1218				IBC2	0xFF30 1218
		dseg		same for next two	
				...	
0xFF30 2000				DBS register	0xFF30 2000
			drseg	D-breakpoint #1 regs	
0xFF30 2100				DBA1	0xFF30 2100
0xFF30 2108				DBM1	0xFF30 2108
0xFF30 2110				DBASID1	0xFF30 2110
0xFF30 2118				DBC1	0xFF30 2118
0xFF30 2120				DBV1	0xFF30 2120
0xFF30 2124				DBVHi1	0xFF30 2124
				D-breakpoint #2 regs	
0xFF30 2200				DBA2	0xFF30 2200
0xFF30 2208				DBM2	0xFF30 2208
0xFF30 2210				DBASID2	0xFF30 2210
0xFF30 2218				DBC2	0xFF30 2218
0xFF30 2220				DBV2	0xFF30 2220
0xFF30 2224				DBVHi2	0xFF30 2224
0xFF30 2228					0xFF30 2228
0xFF3F FFFF					0xFF3F FFFF
0xFF40 0000					0xFF40 0000
0xFFFF FFFF					0xFFFF FFFF

- debug entry: 就是调试异常入口点。因为它位于 dmseg，调试代码可以完全在探测器存储区实现，允许调试一个没有为调试预留物理内存的系统。如果没有探针提供的调试异常处理程序——由探针自己用 **EJTAB_CONTROL(ProbTrap)** 位通知的条件——调试器入口点将会在 `0xBFC0 0480`（在非高速缓存空间异常入口点附近）。

12.1.7 EJTAG 的 CP0 寄存器，尤其是 Debug

在正常情况下（具体说，就是不处于调试模式下），调试单元中软件唯一可见的部分就是一组三个 CP0 寄存器：

- **Debug** 有配置和控制位，下面详细介绍。
- **DEPC** 保存自上次调试异常以来重新开始的地址（由 **deret** 指令自动使用）。
- **DSAVE** 是一个 CP0 寄存器，仅仅是提供 32 位的读写空间。供调试异常使用，以便先保存一个通用寄存器的值，然后再用该寄存器作基址保存其它寄存器。

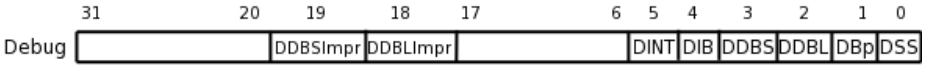

图 12.1: 调试寄存器中异常原因位

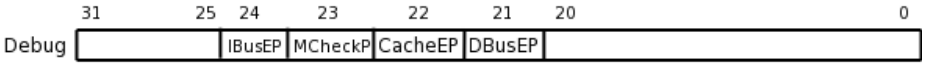

图 12.2: 调试寄存器——异常等待标志位

Debug 是最复杂和最有意思的。它定义了很多位域，我们把它分为三组：图 12.1 中的调试异常原因位，图 12.2 有关正在等待的普通异常（想要发生但是因为处于调试模式而不能）的信息，还有其余一切。"其余一切" 类包括了最重要的位域，如图 12.3 所示。

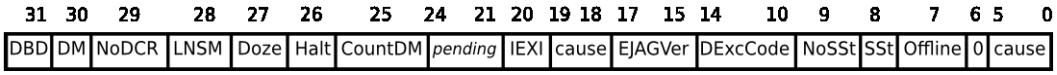

图 12.3: EJTAG 调试寄存器的各个域

图 12.3 中的域（最重要的域）为：

DBD: 告诉你的调试异常发生在一个分支延迟槽中。此时 **DEPC** 指向分支指令，通常就是重新开始的正确地方。

DM: 调试模式——从用户态发生调试异常时置位，由 **deret** 复位。这里不可写。

NoDCR: 只读位——如果实现了 dseg 区则为零。如果该位为 1，那么不清楚到底是那种类型的 EJTAG 单元。

LSNM: 如果你想要 dseg 消失，设置该位为 1，即使在调试模式里也行，以便能访问被覆盖的内存区。这会使得 EJTAG 单元的控制系统大部分不能用，所以可能只在某个特定的 load/store 前后才用。

Doze: 在调试异常之前，CPU 处于某种省电模式。

Halt: 在调试异常之前，CPU 被停止了——可能在 **wait** 指令之后休眠了。

CountDM: 当且仅当调试模式中 **Count** 寄存器继续运行时置 1。有时该位可写，所以你得选择：其它时候为只读位，告诉你 CPU 怎么做的。

IEXI: 置 1 时推后非精确异常。缺省情况下进入调试模式时置 1，离开时清零，但是可写。一旦该位清零，被推迟的异常立刻发生：在那之前，通过察看图 12.2 中的等待位就可以看到发生了的异常。

EJTAGver: 只读位——表明该实现符合规范的哪个版本。已知的合法值为：

0	2.0 及更早的版本
1	2.5 版本
2	2.6 版本
3	3.1 版本

DExcCode: 你从调试模式里刚刚处理的任何非调试异常的原因——第一次进入调试模式时，该域为未定义。可能的取值就是为 **Cause(ExcCode)** 定义的值，列在表 3.2 中。

NoSSt: 只读位——如果没有实现单步操作则为 1。通常都有单步操作，所以该位通常为 0。

SSt: 该位置 1 则使能单步操作。

OffLine: 单线程 CPU 常常不实现该域，这种情况下读出为 0。

如果实现了该位，除了调试模式以外，都可以设置该位为 1 让 CPU（或者多线程 CPU 中的 CPU 线程）停止运行指令。这样设计的目的是允许调试器关闭多处理器/多线程系统中的一部分处理器/线程，同时仍然让这些处理器/线程留在调试模式。只能在调试模式中清零，所以该位为 1 时若执行了 **deret**，那么什么都不会发生，除非是碰到一个外部触发的调试中断。

图 12.1 的各个域为：

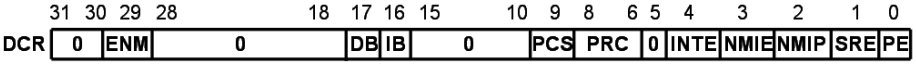

图 12.4: 存储器映射的 DCR（调试控制）寄存器的各个域

DDBSImpr, DDBLImpr: 分别为非精确的store/load断点——参见第12.1.11节。**DEPC** 可能指向比触发断点的 store/load 指令较晚的一条指令。调试器或者用户（或者二者一起）不得不尽最大努力处理。

DINT: 由 EJTAG 引脚上脉冲信号引发的调试中断。

DIB: 碰到了一条指令断点。

DDBS: 碰到了一个精确的存储断点。

DDBL: 碰到了一个精确的加载断点。

DBp: 碰到了任一种类型的调试断点（常和上面之一同时出现）。

DSS: 单步调试异常。

"pending"标志（图 12.2）记录了由调试模式中运行的指令引发但是因为非精确并且 **Debug(IEXI)** 置位而没有发生的异常条件。异常保持等待状态，这些位保持置位，直到 **Debug(IEXI)** 被直接清零或被 **deret** 间接清零，此时异常已经交付，对应的等待位清零。

与不同异常原因联系的单个位：

- IBusEP: 取指时总线出错。
- MCheckP: 机器检查（通常是非法的 TLB 更新）。
- CacheEP: 当从高速缓存读取时奇偶校验码/纠错码出错。
- DBusEP: 有些外部读取报告的总线错。

在任何具体的 CPU 上，这些位其中的某些总是零，因为对应的异常总是精确的（精确异常总是立即处理的）。

12.1.8 DCR（调试控制）存储器映射寄存器

除了 CP0 **Debug** 寄存器之外，还有更多的 EJTAG 控制和标志位于一个叫做 **DCR** 的寄存器，它通过存储器映射到调试专用的 dmseg 内存区位置 `0xFF30 0000`，所以只在 CPU 处于调试模式时才能访问。各个域如图 12.4。

DCR 中的许多域通过 JTAG 可访问的某些寄存器与探针共享。

这些域包括：

ENM: （只读）报告 CPU 尾端（1 == 大尾端）。

DB/IB: （只读）如果 EJTAG 数据/指令硬件断点可用则为 1。很难理解为什么会构建一个没有任何断点的 EJTAG 单元。

PCS, PCR: 如果"PC 取样"特性可用，则 **PCS** 读出为 1。此时，**PCR** 是一个三位的域，定义取样频率为每 2^{5+PCR} 个周期取样一次。详情请参阅第 12.1.12 节。

INTE/NMIE: DCR(INTE) 清零则禁止非调试模式的中断（是一个不同于各种非调试模式可见的中断使能位）。这是考虑到调试器有时想要运行内核例程（或许要找出操作系统相关的信息）但不想由于中断而失去控制。

DCR(NMIE) 屏蔽非调试模式中的不可屏蔽中断（很好的悖论）。

复位后 **DCR(INTE,NMIE)** 都为"1。"

NMIP: （只读）表示有个不可屏蔽中断正在等待，一旦离开调试模式后就会（根据上面的 **DCR(NMIE)**）响应中断。

SRE: 如果实现，该位写入零以防止软复位。

PE: （只读）探针控制的使能位的软件可以读写的版本。当置位时，调试异常重新指向探针控制的 dmseg 内存区。

12.1.9 EJTAG 的断点硬件

EJTAG 断点有监控取指、数据读写的硬件。如果你设置了一个断点，那么如果访问的地址匹配（也可能是数据匹配）就会引发调试异常。

指令和数据断点是分开的。EJTAG 规范允许构建一个没有断点硬件的单元（当然这样的话没多大用处），但是通常都有二到六个断点。断点有下列特性：

- 仅作用于虚拟地址而不是物理地址。但是你可以通过指定要匹配的 ASID 把断点限制到单个地址空间。在操作系统使用多个地址空间的地方，调试器要和操作系统合作才能执行有用的工作。
- 用一个逐位的地址掩码允许一定程度的模糊匹配。
- 断点可以设置成根据读写的数据值而条件触发。但是数据敏感的条件测试可能导致非精确的异常——等到数据可用于匹配时，要防止后续指令生效已经太晚了。如果想更多了解精确和非精确异常，参见第 5.1 节。

有一个全局的指令方和一个数据方控制寄存器，映射到 drseg 区，仅当处于调试模式时在表 12.2 中的地址上可以访问。称其为 **IBS** 和 **DBS**，如图 12.5 所示。

图 12.5 中的域如下：

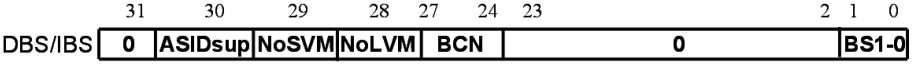

图 12.5: IBS/DBS（EJTAG 断点状态）寄存器的各个域

ASIDsup: 若断点可以用 ASID 匹配来区分不同地址空间的地址则为 1。很难想象任何现代的调试单元没有这个功能。

NoSVM, NoLVM: 若你不能用 load/store 的数据值来限制断点则为 1。仅在 DBS 寄存器中有效。

BCN: 可用的硬件断点数。指令方和数据方的是分开的。

BS1–0: 表示已经匹配的断点的位域。调试软件必须在监测到断点后清除一位。该域的大小取决于指令方或数据方一方提供的断点个数。

每个 EJTAG 硬件断点（"n"为 0-3，选择一个特定断点）通过四到六个分开的寄存器设置:

IBCn, DBCn: 图 12.6 所示的断点控制寄存器。

IBAn, DBAn: 断点地址。

IBAMn, DBAMn: 断点地址比较的掩码屏蔽。掩码中的"1"标记被排除出比较的地址位，所以精确匹配时设置该寄存器为零。

巧妙地让 IBAMm(0) 对应于略有失真的指令地址的位 0，可以用来跟踪 CPU 是否在运行 MIPS16 指令，允许你决定某个指令断点只能在 MIPS16（或者非 MIPS16）模式中发生。

IBASIDn, DBASIDn: 指定一个 8 位的 ASID，可以和当前的 **EntryHi(ASID)** 域比较以过滤筛选断点，这样断点只能发生在位于正确地址空间（典型的对应于一个 Linux 进程）的程序中。对 ASID 的检查可以分别用 **IBCn(ASIDuse)** 和 **DBCn(ASIDuse)** 使能或者禁止——参见图 12.6 及其说明。这些寄存器每一个的高 24 位都总是为 0。

DBVn, DBVHin: 在 load/store 断点中要匹配的值。**DBCHin** 定义 64 位 load/store 匹配的第 31–63 位。真正的 64 位 CPU 的 JTAG 硬件断点有 64 位的 **DBVn** 寄存器，所以不需要 **DBVHin** ——之所以提供该寄存器是因为某些 32 位的 CPU 对于双精度的协处理器（可能是浮点单元）实现了 64 位的 load/store。

注意，你可以通过设置字节通道比较屏蔽 **DBCn(BLM)** 为全 1 来禁止数据匹配（以得到一个纯地址的数据断点）。

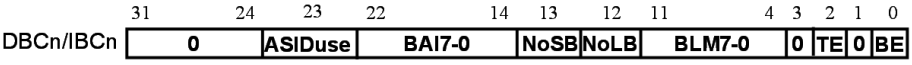

图 12.6: 硬件断点控制（IBCn, DBCn）寄存器的各个域

现在我们来看看图 12.6 中的控制寄存器。

各个域为：

ASIDuse: 设置为 1 的时候同时比较 ASID 和地址。

BAI7–0、**NoSB**、**NoLB**、**BLM7–0** 仅仅适用于数据方的断点：

BAI7–0: 忽略字节存取（byte (lane) access ignore）——听起来挺神秘的。但其实只是一个地址过滤器。

当你设置一个数据断点时，你可能想要在相应数据的任何访问上都中断。通常你并不想让断点变成依赖于被访问的是一个字节、一个字甚至向左对齐的字的条件断点：但是直接设置断点的地址匹配就会是那个效果。

为了确保你能够捕获对某个位置的所有访问，你可以用地址掩码禁止双字以下的地址匹配，然后用 **DBCn(BAI)** 标记双字内部感兴趣的字节：好了，只是为零的位表示感兴趣的字节，为 1 的位表示要忽略的字节（这就是助记符得名的来历）。

DBCn(BAI) 位用 32 位或 64 位数据总线内的字节号编码；所以要小心，数据的字节地址和字节总线编号的关系受尾端的影响。

NoSB, NoLB: 设置为零分别使能[6] store/load 断点。

BLM7–0: 数据比较时的字节掩码。零位代表比较该字节，1 位代表忽略该值。设置该域为全 1 就禁止数据匹配。

其余的域同时适用于指令方和数据方的断点：

TE: 设为 1 作为 PDtrace 指令跟踪的触发器，见第 12.1.3 节。

BE: 设为 1 激活断点。该域复位时为零以免因为寄存器的随机设置而导致虚假的断点：不要忘记对它置位！

12.1.10 理解断点条件

确定一个硬件断点在什么时候检测断点条件及触发异常，涉及到许多不同的域和设置。

在所有情况下，如果你已经处于调试模式就不发生断点...那么要触发断点，需要满足下面的全部条件：

[6] 感觉"1 表示使能" 更加符合逻辑一些。这里用 0 表示使能的好处在于零值代表读写时断点都发生，比起断点"从不发生"更适合做默认值。

1. 断点控制寄存器使能位 **IBAn(BE)/DBAn(BE)** 置位。
2. 程序取指、存取数据产生的地址和断点的地址寄存器 **IBAn/DBAn** 中的位相匹配，**IBAn/DBAn** 相应的地址屏蔽寄存器位为零。
3. 要么 **IBCn(ASIDuse)/DBCn(ASIDuse)** 为零（这样我们就不用关心匹配哪个地址空间），要么当前运行程序的地址空间 ID，即 **EntryHi(ASID)** 等于 **IBASIDn/DBASIDn** 里的值。

有关指令断点的就这些了，但是对于数据方的断点，你需要区分是读还是写，也能根据双字以下地址的区分访问：

4. 如果是加载并且 **DBCn(NoLB)** 为零，或者是存储并且 **DBCn(NoSB)** 为零。
5. 该加载或者存储碰到了双字内部对应 **DBCn(BAI)** 为零的一个字节。

如果你并不想比较 load/store 的值，那么你把 **DBCn(BLM)** 设成全 1 就行了。如果你还想要考虑存取的值，那么还要满足数据比较条件：

6. 加载或者存储的数据按出现在系统总线上的格式，与 **DBVHin** 以 **DBVn** 中的 64 位内容中，那些在 **DBCn(BLM)** 中对应位为零的每一个 8 位组相匹配。

就这些了。相当的复杂，但合乎逻辑。

12.1.11 非精确的调试断点

指令断点和数据断点只是根据地址条件筛选，触发时 **DEPC** 指向匹配的指令。更确切的说，这种异常属于在第 5.1 节讨论的意义下的"精确异常"。

MIPS 体系结构 CPU 的大多数异常都是精确的。但是许多 MIPS CPU 对 load/store 进行优化，允许 CPU 一直运行到要用 load 的数据时候，这样根据数据值筛选的数据断点就是不精确的。当硬件监测到断点匹配时，哪条指令正在运行，调试异常将发生在哪条指令（通常是指令流里更晚的指令）身上。这种情况调试软件必须尽量设法处理。

12.1.12 EJTAG 的 PC 取样

EJTAG 规范和探针的最近版本中有一个很有价值的窍门，PC 取样提供了一种不影响程序的方式来收集正在运行的系统的活动的统计信息。通过察看 **DCR(PCS)** 中适当的位就可以找出你的 CPU 是否提供这个功能。

PC 取样硬件周期性地查看当前 PC 的值，把那个值记录到调试探针可以检索的地方。然后就是由软件构造一段时间内的取样直方图，能够（在统计意义上）让程序员看到 CPU 在什么地方花费了最多的时钟周期。这个功能不仅有用

处，而且大家都熟悉：以前好多年系统用过的就是基于中断的 PC 取样，这种方式对程序的运行有影响。所以有许多工具（例如 GNU 的 gprof）可以直接解释这类数据。

当 PC 取样配置进了 CPU 之后，就会连续运行。甚至当 CPU 停在 **wait** 指令（花费在等待上的时间仍然可能是要测量的时间）时，也不会停止。在一个典型的实现上，你可能会选取每 32 个时钟周期取样一次或者很少情况下选择每 4096 个周期取样一次。既然它连续运行，复位后取样周期缺省为最大值是件好事。

在每个取样点，在该周期完成的指令的地址（若无指令完成，则取下一条将完成的指令的地址）被存放在一个 JTAG 可访问的寄存器中。取样频率由 **DCR** 寄存器中的一个域控制。

硬件不仅存储指令地址的 32 位，而且存储当前的 ASID（这样可以解释虚拟 PC）还有一个永远写入 1 的"新"位，探针可以用它避免对同一个取样重复计数。

12.1.13 无接探器使用 EJTAG

EJTAG 单元可以由一个完全运行在目标系统上的传统调试器使用，特别是，可以用来构建操作系统核心内置的调试功能。但是因为发明 EJTAG 主要是为了给接探器即探针使用，这样做有几个问题需要处理：

- 处理调试异常：没有接入探针并使能时，调试异常入口点位于不经高速缓存的 kseg1 的 ROM 区的 `0xBFC0 0480` 地址（如果需要检查是否有使能的探针，可以读取 **DCR(PE)**）。你要把代码放到该入口点，然后将控制转移到你的调试异常处理程序，该异常处理可能做进了操作系统的内核。

- 在内核中通过调用进入调试模式：你不能直接切换到调试模式：只能通过调试异常进入。但是你需要进入调试模式才能看到 dmseg 存储区，以便可以存取断点控制寄存器一类的东西。所以对于纯软件的调试器，你需要一种方法来调用调试模式的调试器支持例程。

  这就需要"系统调用"的一种调试异常版本，采用植入 **sdbbp** 指令实现。调试异常处理程序需要区分这些调试器系统调用和其它调试异常——最可能的做法就是通过查看 **DEPC** 中的返回地址来区分。调试器系统调用参数可以放到通用寄存器里。

  类似的，你也不能直接切换出调试模式——要离开调试模式，你需要执行 **deret** 指令。

  在有些情况下，你可能想要在通常的核心态执行调试器的大部分：所以在因断点匹配或调试指令而短暂进入调试模式之后，调试模式软件可以修补正常的 CP0 寄存器以模拟一个更类似传统方式的异常。

- 仅适用于虚拟地址的断点：注意（跟非 EJTAG 的 CP0 观察点不同）所有的 EJTAG 断点都工作于虚拟地址，可以选择加上地址空间 ID (ASID) 进行限制。

- 从异常模式或者其它"非法"的地方处理调试断点：乍一看上去象是个大问题。调试断点是属于异常，但是因为它们属于"超级异常"，所以并不按照内核其它部分的规则出牌。但其实这和我们前面讨论的是一个问题。很可能一个利用 EJTAG 的本地机调试器程序大多数时候并不运行于调试异常模式。所以调试异常处理程序仅在保存状态、然后在安全的地方存储返回地址、以及用一个人为的 **deret** 落入正常的高特权级（核心）态期间，运行于调试态。

  当然有时候，调试异常入口点可能来自于某个危险的地方——最明显的就是来自已经运行于普通异常模式的代码。接到这种异常的调试器不得不极为小心，一般不能简单地调用通常的核心态例程。调试器作者需要计算他们在多大程度上支持途经异常的调试。

12.2 "EJTAG"之前的调试支持——断点指令和 CP0 观察点

MIPS 体系结构在 EJTAG 调试单元出现之前已经存在了许多年，所以还有一些传统的调试功能。这些包括 **break** 指令，这个指令只是简单导致一个异常，同时拥有很多未解释的位，可由调试软件赋予其意义。

但是许多 CPU 同时也实现了多达四个的硬件观察点，由几个 CP0 寄存器控制。每个观察点指定一个虚拟地址，可以用于在每次取指、存储和加载操作中检查，如果地址匹配就会引发异常。

并非所有的调试器都用到硬件观察点；有些可能完全不用，另一些使用 EJTAG 功能。

与 EJTAG 断点相比，观察点不能进行模糊地址匹配：从来不是数据敏感的；CP0 观察点可以工作于指令方或者数据方（有些标志可以控制具体观察点属于那边）。当然了，观察点触发后只是引起普通异常，所以不能用于调试异常处理程序。

当 CPU 已经处于异常状态时（当 **SR(EXL)** ——或者等效的错误或调试异常位——已经置位），仍然可以匹配观察点条件。此时处理异常会导致无用的混乱，所以异常被推迟到后面。想要发生异常的事实记录在 **Cause(WP)** 位，当 CPU 返回正常操作时观察点异常就会发生。

观察点寄存器如图 12.7 所示。

观察点地址仅维护到最接近的双字（八字节），所以只有地址位 3–31 参与匹配。在 64 位 CPU 上，WatchLo 增长到 64 位以便能够定义一个完整的虚拟地

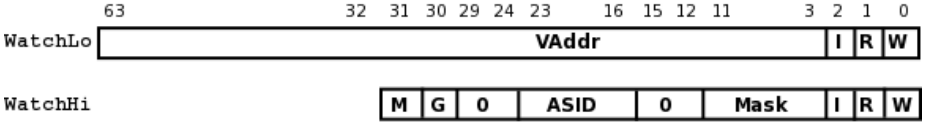

图 12.7: WatchLo 和 WatchHi 寄存器的各个域

址。

其它的寄存器域如下：

WatchLo(I,R,W): 使能观察点检查——若 WatchLo(I) = 1 检查取指，若 WatchLo(R) = 1 检查读取，若 WatchLo(W) = 1 检查写入操作。这些位的任意组合都是合法的。

WatchHi(M): 只读的接续位——若为 1 表明至少还有一对观察点寄存器。

WatchHi(G): 全局的——该位设置为 1 则匹配时不考虑 ASID，见下。

WatchHi(ASID): 设置该域（WatchHi(G) 为零）只匹配本地址空间的访问。使用 MIPS TLB 的操作系统在 **EntryHi(ASID)** 里保存地址空间，如第 6 章所述。

WatchHi(Mask): 该域为 1 的位表示 **WatchLo(VAddr)** 中相应的位可以不用理会，允许你捕获对齐到从 8 到 2048 之间的 2 的幂大小的内存区的访问。

WatchHi(I,R,W): 观察点异常后读取这些域找出异常是由于取指还是读或者写操作引起的。软件要清除这些位才能再次使用，这些位的实现采用的是“写 1 清 0 ”——就是说，对 **WatchHi(R)** 清零的时候向 **WatchHi** 写入一个位 1 上为 1 的值。这样做可以让你将从 **WatchHi** 读来的值直接再写回去就方便地实现清零。

12.3 PDtrace

PDtrace 是（本章前面介绍的）EJTAG 调试单元的附件，可以跟踪程序执行用于以后的重构。执行跟踪可以保存在片内的存储器中或者采用一些略微奇特的高速信令技术实时回放到探针。

最早的跟踪工具只记录执行地址（PC），但是 PDtrace 系统也能跟踪读写地址甚至读写的数据值。

跟踪执行并不需要很多数据。假定分析程序拥有系统的完整的二进制代码，所以当顺序执行时只需要知道 CPU 顺序走了多远；当遇到条件分支时，只需要知道条件是否满足。但是对于寄存器跳转之类的指令，跟踪需要记录完整的地址。

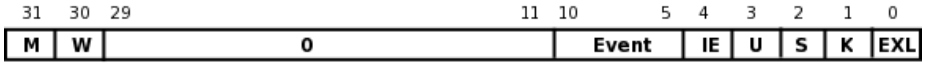

图 12.8: PerfCtl 寄存器的各个域

跟踪信息记录到片上的存储器虽然简单快捷；但是由于实际的片上跟踪缓冲器都很小，通常只能记录程序运行的一小段时间。

当把跟踪信息送到到探针时，可能有大量的跟踪存储器（几十甚至几百兆字节）。这要求封装时有几根引脚连接跟踪数据线。当跟踪数据生成的太快以至连接速度跟不上（这种可能性很大）时，你要么降低 CPU 速度直到连接速度跟上，要么丢弃一些跟踪数据。

EJTAG 断点被复用于提供细粒度的跟踪控制，当遇到特定断点时动态切断或接通跟踪数据流。

几乎只有探针供应商对 PDtrace 感兴趣，所以这里不多讲了。其手册可以从 MIPS 公司得到（`www.mips.com`）。

12.4 性能计数器

性能计数器在那里让软件和硬件工程师为了性能调优（偶尔为了调试）而找出更多有关系统行为的信息。每个计数器在你的具体 CPU 选定实现的若干事件之一发生时递增。

现代的典型 CPU 拥有两个性能计数器。大多数可以对常见的有用事件计数：消逝的时间周期、完成的指令、高速缓存未命中等等。心里要记住一点，CPU 设计本来就难以制造和验证，增加的性能计数器之类的“外围”的功能进行计数的事件，极可能就是那些容易计数的事件。容易计数的事件并不一定恰好让软件工程师看起来那么自然，所以要持小心谨慎和怀疑的态度对待事件描述。在你真正确信是在对某个事件进行计数之前，合理的做法是在受控条件下对计数器的行为进行实验。

每个 32 位计数器都伴有一个控制寄存器，如图 12.8 所示，其中：

M: 接续位；若为 1 表示后面还有一对性能计数器控制/计数寄存器。

W: 如果计数寄存器是 64 位则为 1。一般不太可能。

Event: 决定对什么事件进行计数——这个表完全取决于你的 CPU，你需要阅读相关的手册。

IE: 若置位则当计数器“溢出”到第 31 位时引发中断。这可以用来实现扩展的计数，但是更多地用于（要事先适当预设计数器）在事件发生了一定次数之后通知软件。

U, S, K, EXL: 分别对用户态、管理态、核心态和异常态（即 **SR(EXL)** 置位）的事件进行计数。你可以设置成这些位的任意组合。

第 13 章　天上掉下个林妹妹——GNU/Linux

为什么一本关于计算机体系结构的书，一直讲着体系结构，冷不丁一下子连续几章内容都转去讲某个具体的操作系统呢？真让人有点摸不着头脑，感觉象半路杀出个程咬金；继续读下去，你就会发现，其实应该说感觉真象是“天上掉下个林妹妹!” 她的名字就是林纳（克）斯(Linux)。

请听我慢慢道来。CPU 就是跑操作系统用的。一个 CPU “体系结构”就是描述一个有用的 CPU 的行为的，而一个有用的 CPU 是要在操作系统的控制下运行程序的。尽管运行于 MIPS 的操作系统很多，Linux 由于其公开性而具有独特的优势。任何人都可以下载其源代码，任何人都可以了解它是怎样工作的。

任何操作系统都不过是一些程序的集合，一些聪明的程序——相对大多数软件而言——基于多年研究提炼出的思想之上构造起来的程序的集合。操作系统应当特别地稳定（不死机）和安全（不让某些程序干未经操作系统授权的事情）。

对“Linux”这个名字正确的用法是用来仅仅指一个操作系统的内核，该操作系统最开始由 Linus Torvalds 开发，其随后的历史，怎么说呢，还就是历史。系统的其余部分（要比内核大得多）大多数来自于自由软件基金会（Free Software Foundation）在 GNU 旗下组织的项目。大家现在有时忘记了这一点，把整个系统都叫做“Linux”。

这两部分最早都是源于在 1970 年代由贝尔实验室开发的 Unix 系统。也许是因为贝尔实验室觉得它没有商业价值，在当时以史无前例的“开放” 方式向各个学术机构广泛分发其软件。但是它并不是“开放源码的” ——许多程序员在大学里为 Unix 工作，结果发现他们的贡献要么丢失了，要么被贝尔实验室（及其继承者）拥有。这个过程所带来的挫折最终激发人们来写一个“真正自由”的替代品。

最后剩下的关键部分就是内核。内核属于非常难写的程序，但其所以拖到最后的原因源于文化方面：操作系统的内核被公认为属于学术研究的范畴，而学术界希望能够超越 Unix，而不是重新创建 Unix。后 Unix 的潮流是倾向于一个小型、模块化的操作系统，由明确分开的部件构建而成；但在这个基础上构建的

操作系统从来没有得到过广泛的用户群体支持。[1]

Linux 的胜出是因为它是一个更加注重实用的项目。Linus 以及他的开发伙伴想要一个能够（首先是在 x86 桌面系统上）用的东西。当 Linux 内核与来自最终成为自由软件的 BSD 4.4 的后代竞争的时候，BSD 的支持者坚决认为严格的项目组织管理必不可少。但是 Linux 社区达成的有关开发风格的共识，远比 BSD 他们要开放得多的。Linux 发展极为迅速。有时候 Linux 的快速发展也是因为 Linux 人员非常乐于接纳 BSD 代码。不久以后 Linux 就胜出了，其组织管理也随之改善。

13.1 基本概念

要想理解某件制品，你需要能够很好理解其领域的专家使用的一些术语的特定意义。你特别容易被你已经知道但是具有不同意思的术语搞糊涂。Unix/Linux 的历史源远流长，形成了很多具有特殊意义的词。

- 线程：我所知道的“线程”最通用的定义就是“按照程序员指定的次序运行的一组计算机指令。”Linux 内核有明确的线程概念（对于每个线程，有一个 `struct thread_struct`）。基本上是同一个东西，只是按照我的定义，底层的中断处理程序是一个恰好借用了被中断线程的环境来运行的特殊线程。两个定义都有价值，必要时我们用“Linux 线程”这个词加以区分。

  Linux 热爱线程（目前我正在打字的台式机上有 134 个线程）。大多数这些线程对应于一个活动的程序——但是还有好几个用于特殊目的的线程仅运行于内核，并且有些应用程序拥有多个线程。

  内核的基本工作之一就是调度 ——下次选择哪个 Linux 线程运行。参见下文关于调度器的内容。

- 文件：一块有名字的数据。在 GNU/Linux 里，程序和自身进程以外的世界打交道都是通过读写文件完成的。文件就是你写入数据以后能够读出的东西。但是也有连接到设备驱动程序的特殊文件：从其中一个读取则数据来源于键盘，对另一个写入则你的数据被解释为数字音频送到扬声器。Linux 内核倾向于避免太多新的系统调用，还提供了让应用程序获取有关内核信息的特殊的 `/proc` 文件。

- 用户态和系统调用：Linux 应用程序运行于用户态，即 MIPS CPU 的低特权级状态。在用户态，软件不能直接访问内核所处的地址空间，它能寻址的所有位置都映射到内核同意应用程序使用的地方。在用户态，你不能运行协处理器零的 CPU 控制指令。

[1] 有些人声称 Windows/NT ——从而 Microsoft Windows 的大多数现代版本——拥有微内核。那也许是真的，但是在它走向统治世界的路上，显然失去了小型化或者模块化的特性。

GNU/Linux 运行于用户态的应用程序代码常常叫做*用户方(userland)*。

要从内核获得任何服务（最常见就是读写文件），应用程序都要执行系统调用。系统调用是特意植入的异常，由内核的异常处理程序解释。异常时切换到高特权级模式。

通过系统调用，Linux 应用程序线程和处于高特权级模式的内核（当然在这里运行的是信任代码）能够顺畅运行。系统调用结束后，从异常代码的返回涉及到一条 **eret**，该指令确保同时完成切换回用户态和返回用户态代码。

- 中断上下文：Linux 尽量避免禁止中断。当 Linux 运行时，在任一时刻，一个 CPU 上有一个活动的线程：[2] 所以中断在完成任务返回前，一直借用看上去属于该线程的环境。中断处理程序调用的代码处在中断环境，有许多事情这种代码都不能做。例如，不能做任何可能需要等待其它软件活动的事情。如果你的键盘输入例程要把所有击键记录到一个文件中[3]，那么你不能从中断处理程序中调用文件输出。

  做到这一点有更好的办法：例如，你可以让键盘中断安排唤醒一个线程来获取击键并记录。

- 中断服务例程（ISR）：设备驱动程序中最底层的中断代码通常叫做 ISR。在 Linux 中鼓励大家保持这些代码简短：如果有许多工作要做，可以考虑用下一章介绍的一种“下半部(bottom half)”。

- 调度器：一个内核例程。操作系统维护一个就绪线程（比方说，没有因为未完成的 I/O 传输而阻塞）列表，该表按照优先级排序。优先级是动态的，周期性的重新计算——主要是为了保证长时间运行的计算不会过多占用 CPU 以至于无法响应事件。应用程序可以降低自己的优先级主动进入后台，但是通常不能提高自己的优先级。

  在任何中断处理之后，都会调用调度器。如果调度器发现另一个线程更应该得到运行机会，就停止当前线程运行优胜者。

  老的 Linux 内核不是*抢占式的(preemptive)*：一旦一个线程在内核态运行，就让它一直运行到要么自愿请求重新调度（因为等待某个东西），要么控制传回用户态——只有到那时，内核才会考虑线程切换。

  非抢占式内核容易编写。你写的内核代码序列可能要考虑正在运行到半空时被中断处理程序意外打断，但是你知道，不会在你运行到半中腰的时候被其它主流内核代码意外打断。但是非抢占式内核导致了过度延迟，降低了响应性。

[2] 就算内核在关机模式中等待，也有一个在执行 wait 指令的线程。
[3] 从安全的角度看也许不是一个好主意，但是仍然...

当你有一个 SMP 内核（此时两个 CPU 同时执行同一个内核）时，由并行线程相互作用所带来的巨大的自由度就丧失了。要让 SMP 内核顺利运行，需要跟踪几百个可能的线程间相互作用并用适当的锁进行保护。SMP 锁（几乎在所有情况下）所在的地方，恰好就需要允许调度器停止一个正在运行的内核线程而运行另一个线程：这个叫做“内核抢占”。

现在内核编程的一个重要训练科目就是找出代码序列中那些必须临时禁止抢占的地方。标记这种代码起始和结束的宏的定义会根据内核配置而变化，以便在单处理器和 SMP 多处理器系统上都能正确运行。

- 存储器映射/地址空间：可用的存储器地址到具体 Linux 线程的映射。线程的地址空间通过线程指向的 `mm_struct` 来定义。

  对于移植到 MIPS 体系结构上的 Linux 操作系统（此后简称“Linux/MIPS”），在 32 位处理器上，地址空间的高半部（第 31 位为 1 的地址）只能在内核特权模式下读写。内核代码/数据通常位于称为 kseg0 的这部分的一个角落，这意味着内核本身不依赖于 TLB 的地址转换。

  地址空间的用户部分对于每个用户有不同的映射——只有那些在显式的多线程应用程序中协作的线程才共享用户地址空间（即它们指向同一个 `mm_struct`）。但所有的 Linux 线程都共享同一个内核地址映射。

  传统的运行于单线程应用程序中的线程在一个不同于所有其它线程的地址空间运行，这正是老式的 Unix 之类的系统所称的“进程(process)。”

  在任意给定的时间，应用程序的地址空间的大部分实际上可能没有映射，甚至在内存中根本没有数据表示。试图存取这部分地址会导致一个 TLB 异常，异常由操作系统处理，异常处理程序加载缺失的数据并在返回程序前设置适当的映射。这当然就是，*虚拟存储器*。

- 线程组：位于同一个存储器映射的线程的集合叫做一个线程组。当一个线程组有两个或更多成员时，这些线程相互合作来运行同一个程序。线程组在 Linux 中大体相当于老的 Unix 系统中所谓的“进程”。

- 高位存储器：512 M 以上的物理存储器（不管真的是可读写的内存还是存储器映射的 I/O 地址）不能直接通过 kseg0（经过高速缓存）和 kseg1（不经高速缓存）窗口访问。在 32 位 CPU 上，512 M 以上的物理存储器是 Linux 意义上的“高位内存”，只能通过 TLB 映射访问。在 MIPS CPU 上，你可以通过定义不被替换出去的“禁锢的(wired)” TLB 入口，创建一个永久的映射。但是 Linux 尽量避免使用很快就用完的资源，所以主流的内核代码中避免使用禁锢的 TLB 入口。对于 Linux/MIPS，高位内存映射通过即时创建 TLB 项入口来动态维护。

- 程序库和应用程序：很久以前，在 Unix 之类的系统上的程序都是整块的代

码，运行时必须全部加载。通过对源代码进行编译并与一些库函数粘连在一起来构建程序。库函数就是由工具链提供的预先构建好的二进制代码。

但是这样做有两个问题。一个是库代码常常比用它的应用程序还大，所有程序的体积都膨胀了。另一个就是如果供应商修正了库函数中的一个缺陷，除非每个软件维护人员都重新编译自己的程序，否则你不能从这个修正中充分获益。

替代的方法是，应用程序构建时不带库函数。缺失的库的名字记录进应用程序，这样加载器可以找到必须的库并在程序加载的时候将二者缝在一起。只要该库继续提供同样的函数，一切都没有问题（有个库的版本跟踪系统将会允许库的功能也逐渐演化，但是那已超出我们的范围）。

这样做要付出点代价。在加载时你要从各个片断（每个片断可能是单独更新的）链接成整个程序，而每个部分的确切地址在编译时是不能预先知道的。你无法事先预知哪个位置可用于加载某个具体库。运行时加载器只能做到将每个库加载到下一个可用的空间，这样连库的起始位置都无法预测。一个库函数的二进制代码必须是*位置无关代码(position independent code)* PIC ——不管其代码和数据定位在虚拟地址空间的什么位置都能够正确运行。

我们将在第 16 章讨论 PIC 代码和应用程序地址空间的构造。

13.2 内核的分层结构

从某种观点来看，内核就是通过异常处理程序调用的例程集合。MIPS CPU 上原本的异常发生后进入的“异常模式”环境极为强大而且开销很小但是程序很不好写。所以在内核的每个入口，你要有个类似缩短的启动过程，此时每个“层”构造下一层必要的环境。此外，当你从内核退出时，你以相反的顺序再次经过同样的层，在最后的 **eret** 指令返回到用户程序之前短暂地穿过异常模式。

内核的各种不同环境都是多少经过精心构造的软件组成的，这在一定程度上弥补了异常处理环境的一些局限。让我们按照进入内核时自底向上的顺序列出几点：

13.2.1 异常模式中的 MIPS CPU

响应异常之后，紧接着 CPU 立即设置 **SR(EXL)** ——进入异常模式。不论其它的 **SR** 的设置如何，异常模式强制 CPU 进入内核特权模式并关闭中断。还有一点，除了一种极特殊的方式之外[4]，CPU 在异常模式中不能响应嵌套的异常，。

[4]MIPS 历史上有些隐秘的窍门利用异常模式中的特殊异常行为——但是 Linux 不用这些。

异常处理程序的最初几条指令通常是保存 CPU 通用寄存器的值，其中的值可能对于异常发生之前的软件有重要意义。它们被保存在中断发生时正在运行的进程的核心栈上。MIPS 的本性要求保存寄存器的存储操作至少要先用一个通用寄存器，这就是为什么名为 **k0** 和 **k1** 的寄存器被保留给异常处理程序使用的原因。

异常处理程序也要保存某些关键的 CP0 寄存器：**SR** 在异常处理程序的下一步会被修改，但是当我们返回时应当保持整个 **SR** 的值和刚进入异常时相同。一旦完成这一步，我们就可以通过修改 **SR** 离开异常模式，但是我们依然关闭中断。

象 x86 之类的 CISC CPU 没有等效的异常模式；MIPS 异常模式中的工作在 x86 中由硬件完成（实际上是看不见的微代码完成）。x86 到达中断或自陷处理程序的时候寄存器已经被保存了。

运行于 MIPS 异常模式的软件可以看作生成了一个虚拟机，由该虚拟机来负责在紧接异常之后保存被中断的用户程序的状态，以及负责为后来返回的 **eret** 指令作准备时恢复这些状态。程序员对于在异常模式中的行为要特别谨慎。异常模式基本上位于保证内核线程安全的软件锁的控制之外，所以异常代码只能极为谨慎地与内核其余部分交互。

在实现系统调用的异常这个特例中，根本不需要真正保存通用寄存器（只要异常处理程序不写入 **s0**–**s8** 的“保存(saved)”寄存器即可）。在系统调用或任何其它非中断的异常中，你可以直接调用到运行于线程环境的代码。

有些特别简单的异常处理程序从不离开异常模式。这种代码甚至不必保存寄存器（只要避免使用多数寄存器）。一个例子就是第 14.4.8 节所述的“TLB 重填”异常处理程序。

也有可能——虽然现在很少见了——让一个中断处理程序短暂运行于异常级，完成最少的工作后返回。但是这样的中断处理程序在操作系统级不是真正可见的，必须要在某一点引发一个 Linux 可识别的中断，以便让高层软件处理其数据。

13.2.2 关闭部分或全部中断的 MIPS CPU

如我们在下一章所看到的，中断例程退出异常模式后但还继续运行时至少关闭部分中断。

关闭所有中断运行是一个昂贵但是有效的保证单 CPU 不可抢占的方法（软件处于中断关闭期间花费的最长时间决定在最坏情形下的中断延迟，每个有实时约束的设备驱动程序都要把这点考虑进去）。当然在有第二个 CPU 的情况下关闭所有中断并不能防止重入。

最简单、最短小的 ISR 可以选择不重新使能中断而一直运行到结束——Linux 可以支持这一点，称之为*快速中断处理程序*。当注册 ISR 时设置 `SA_INTERRUPT` 可以得到这个行为。但是大多数 ISR 要在使能高优先级中断的情况下运行一段

时间。

有可能得到一摞嵌套的中断别的中断的中断。无限的递归（意味着堆栈溢出并最终不可避免的崩溃）不会发生，因为 Linux 保证在每个不同的中断优先级最多只能有一个中断进栈。每级保存的数据必须少到最大的中断嵌套保存的信息不会超出一个线程的核心栈空间。

13.2.3 中断环境

中断发生后，即使在中断处理程序重新允许大多数中断并且建立起了完整的 C 运行环境之后，中断代码依然要受到限制，因为它借用了正好被打断的线程的状态（及其核心栈）。

当然进行中断服务是别人的工作，与中断发生时正在执行的线程之间并没有有必然的联系。中断借用了被打断的受害线程的核心栈并且寄生在该线程的环境中运行。软件处于中断环境，为了防止不必要的崩溃，中断环境代码的行为要受到限制。

内核完成的一个关键任务就是调度器，由它决定下一次应当运行哪个线程。调度器是一个例程，由线程调用；有些情况下由处于中断环境的线程调用。一旦中断处理程序的中断环境部分可以到达这一步，让硬件当前的需求得到满足，就可以（也常常）调度一个线程完成中断处理工作，这次是在线程环境下。

13.2.4 在线程环境中执行内核

在线程环境下进入内核，要么是应用程序主动执行了系统调用，要么是其被动地调用了虚拟存储器异常的资源（其低层执行了系统调用或者发生了 VM 异常），要么是由于重新调度的结果——这反过来又是由于中断或者另一个线程因等待事件而主动要求重新调度引起的。

系统调用是一种“带有安全权限检查的子程序调用。”但是其它的异常——特别是虚拟存储器维护异常——基本上也是一样的，当然应用程序在发生异常之前不知道这种特殊的系统调用是不是一定会发生。

并非每个线程都是应用程序线程。不附着于特定应用程序的特殊线程可以用来在核心态的进程环境下为设备管理和其它内核功能调度工作。

线程环境是内核的“正常”态，许多功夫都花在了保证大多数内核执行时间处于这个模式。比如，中断处理程序中被调度到工作队列(work queue)（参见第 14.1 节）的“下半部”代码，就处于线程环境。

第 14 章　软硬件怎样协同工作

让我给你讲一个故事...

嗯，其实应该说是让我给你讲好几个故事，好好讲讲 MIPS 硬件是怎样提供支持 Linux 内核的底层特性的。

- 中断的生命和时间：当某块硬件向 CPU 示意需要照顾时会发生什么？如果中断调度的代码在另一段代码的某个大型操作执行到半路时将其打断，这时中断有可能导致一团糟；有鉴于此，有必要专门花一节讲讲线程、临界区和原子性。
- 系统调用时发生什么：当用户态应用程序调用内核例程时会发生什么，怎样保证安全性？
- Linux/MIPS 中地址是怎样转换的：一讲到虚拟存储以及 MIPS 硬件怎样服务 Linux 存储器地址映射的故事，说来就话长了。

14.1　中断处理的过程和时间

一切都开始于现实世界中发生的某件事：也许你按下了键盘上的一个键。

设备控制器硬件接到数据后激活中断信号。就这样上路了——也许沿途经过 CPU 外部的几处硬件，具体经过哪些硬件情况各不相同，也不是我们现在感兴趣的——一直到它以激活 CPU 的中断信号之一的形式出现，最常见的就是 *Init0-5** 之一。

CPU 硬件在每个时钟周期轮询这些输入。Linux 内核有时候禁止中断，但不常见；更多的情形下 CPU 都可以接收中断。CPU 的响应就是在下次取指的时候接受一个中断异常。

有时很难知道这个过程有多快。一个 500-MHz 的 MIPS CPU，碰到一个中断时，在 3 到 4 个周期内取出异常处理程序（如果位于高速缓存内）的第一条指令：这是 8 纳秒。有经验的底层程序员知道在任何系统中都不能指望只有 8 纳秒的时延，所以下面我们讨论一下到底是什么挡路了。

当 CPU 从异常入口点取指令的时候，同时也置位异常状态位 **SR(EXL)**，这将使 CPU 忽略后续中断并进入内核特权级。控制将转到通用异常入口点：位

于 0x8000 0180。[1]

通用异常处理程序检查 **Cause** 寄存器特别是 **Cause(ExcCode)** 域：零值表示这是一个中断异常。**Cause(IP7–2)** 域表示哪个中断输入线是活动的，**SR(IM7–2)** 表示响应哪些中断。应当至少有一个活动的、使能的中断；软件计算一个对应于活动中断线编号的数，就是 Linux 的 `irq` 号。[2]

在调用更加一般的代码之前，主异常处理程序要保存所有的整数寄存器值[3]——它们属于刚才被打断的线程，在将来某个地方允许该线程继续运行之前必须恢复。这些值保存在中断发生时运行的线程的核心栈里，并相应设置栈指针。有些关键的 CP0 寄存器值也要保存；包括 **SR**、**EPC** 和 **Cause**。

SR(EXL) 置位期间，我们并没有真正准备好调用主内核的例程。在这种状态，我们不能接受另外的异常[4]，比如说，我们不得不小心绕开任何映射的地址。

那么现在我们已经保存了寄存器并设置好了栈，我们可以修改 **SR**，清零 **SR(EXL)** 同时也清零 **SR(IE)** 以免再次发生异常——毕竟我们响应的中断信号仍然活动着。

现在我们一切就绪，可以调用 `do_IRQ()`。这是一个虽然依赖于具体机器但是用 C 写的例程，具有一个相当标准的流程。一个充满寄存器值的结构体作为参数传递给 `do_IRQ()`。该函数的工作就是有选择地禁止特定中断，并且向需要确认的中断管理硬件通知说该中断已经处理。

假定有一个设备 ISR 注册了这个 irq 号，`do_IRQ()` 调用 `handle_IRQ_event()`。不应当长时间关闭全部中断运行；外面有些设备的中断可能需要快速响应。但是我们现在处于一个区域，这里一切都取决于我们处理中断的设备的性质。如果已知该处理程序非常简短高效（`SA_INTERRUPT` 置位），我们可以保持全部中断关闭一直执行下去；如果花费时间较长，那么我们一般要重新使能中断——`do_IRQ()` 已经关闭了我们正在处理的中断。

同一个 irq 号上可以注册多个中断处理程序；这时依次调用每一个。

一旦中断处理程序结束，`handle_IRQ_event()` 再次禁止中断然后返回（这就是 `do_IRQ()` 期望的）。经过一些与具体机器相关的整理工作之后，调用 `ret_from_intr()` 清理中断异常。

在最后从中断返回之前，我们检查在 ISR 期间是否发生了要求重新调度的事件。Linux 有个全局的标志 `needs_resched` 用于这个目的；相关的代码不是那么简单，因为从核心态中断（核心抢占）的线程中重新调度可能导致问题。其实，如果被中断的线程正以核心特权级运行——从保存的 SR(KSU) 值可以看到——全局的 `preempt_count` 非零时我们就不用重新调度。

[1] 当然没有那么简单；现在有些 MIPS CPU 提供专用的中断异常入口点，有些甚至根据是哪个中断信号而转往不同的入口点。但是我们只考虑简单情形——许多操作系统也是这样。

[2] 在许多情况下，异常处理程序可以根据具体系统的外部寄存器获得有关中断号的进一步信息。

[3] 某些 MIPS CPU 对很底层的中断处理程序可以用第 5.8.6 节所述的影子寄存器避免这个开销。

[4] 这并非不可避免。MIPS 体系结构有一些精心设计的地方可以允许在严格控制条件下的嵌套异常。但是 Linux 并未使用。

如果系统并不调度另一个线程，我们只是接下去从中断返回。我们恢复所有保存的寄存器值。特别是，我们要把 **SR** 恢复为刚发生异常后的值，这样又重新返回异常模式，再恢复 **EPC**，它里面保存着重新开始的地址。然后只要运行一条 MIPS **eret** 指令。

如果系统确实调度了另外一个线程，那么被中断的线程就保持原状停在那里，在从中断返回的边缘摇摆不定。当中断受害线程被再次选择运行时，就会一口气冲出来，回到中断时正在做的事情。

有些设备的驱动程序只是从设备取得一点数据然后送到上一级；其它设备的可能要在中断时做许多处理。但是 ISR 不应当在中断环境中运行太久。即使在我们重新使能其它中断之后，设备驱动程序已无条件得到了 CPU 的关注，没有给操作系统调度器插手实施其策略的机会。在 ISR 中执行的工作如果有超过几十条指令的地方，最好把中断处理一直推迟到调度器可以确定是否有更重要的事情要做之后。

Linux 传统上把 ISR 分成“上半部，”在中断时运行，和“下半部，”推迟到重新调度之后运行。[5] 现在的 Linux 内核有个系统叫做*softirq*，可以安排运行 32 个功能之一；几个要求非常高的设备有它们自己的 softirq 下半部，但是大多数共享同一个在 softirq 之上构建的更加灵活的系统，称为 *tasklet*。

二者都是用机器无关的代码实现的，都是在真正的 ISR 返回之后尽快安排辅助中断处理程序运行的方法。

tasklet 最终是由中断处理程序调用的，不能执行任何可能导致无关的中断受害线程睡眠的操作。tasklet 代码不能做任何需要等待的事情。处理这种事情总有不止一种方法。ISR 也可以把额外的驱动程序任务放到工作队列*(work queue)* 中去。工作队列有一个常规的内核线程服务，其函数可以执行任何操作，包括可能导致线程睡眠的操作。

既然这一切都是机器无关的，我们这里就不详细讲了；在文献 Love 2004 中有专门论述。

14.1.1 高性能中断处理和 Linux

与那些轻量级的操作系统相比，Linux ISR 中有相当一部分额外的开销。Linux 无条件的保存所有寄存器并进行几级函数调用（例如，在 `do_IRQ()` 中隔离中断控制器处理）。有可能在 MIPS 中断处理程序中不离开异常模式（即保持 **SR(EXL)** 置位）完成有用的工作——但是传统的 Linux 并不直接支持这种做法。构建一个仔细设计的 MIPS 操作系统使得全部中断关闭的时间最多不超过几条指令是有可能的，但是 Linux 依赖于关闭 CPU 上的全部中断来避免在不适当的地方发生中断。

[5] 名字可能容易混淆。Linux 以前有一个子系统称作“下半部（bottom half）”其函数名字都以“`bh`”打头，但是这已经过时，最后从 2.6 内核中去掉了。这里我们仍然用“下半部”泛指移出中断处理程序之外的工作。

外面有一些人努力改进 Linux 对中断的响应性。虽然降低最坏情况下的中断延迟——从硬件中断信号出现到进入 ISR 之间的时间——很重要，但这不是全部工作。要提高响应性，系统对于输入输出有关的线程调度也要快。

其中一个项目（“低延迟补丁”项目）是渐进的，目标是通过对内核加锁/抢占控制机制进行耐心持续的精化来改进平均性能。其中一些低延迟工作已经进入了主流的 2.6+ Linux 内核。

其余的项目（各种“实时”Linux）则更为激进，通常是把 Linux OS 架空让它根本看不到中断。中断由一种实时操作系统在地下处理，把整个 Linux 仅仅作为实时操作系统的一个线程。当 Linux 禁止中断时，并没有真正禁止硬件中断；只是不让 RTOS 层调度 Linux ISR。

值得指出的是，这两种方法是互为补充的——完全有理由寻求二者兼顾。

14.2 线程、临界区和原子性

Linux 的线程必须意识到它正在操作的数据有可能被其它并发的活动所注意。实现多任务操作系统碰到的主要困难之一就是找出并保护所有这种序列。并发的活动可能来自于：

- 另一个 CPU 运行的代码：只出现在真正的多处理器系统中
- 中断环境下运行的代码：作为中断后果而运行在同一个 CPU 上的代码
- 抢占该线程的另一个线程运行的代码：这点更为微妙，这是由某个别的内核线程在调度（无疑是某个中断的结果）中击败了我们的线程而运行的代码。

当一系列对数据的操作全部完成之后能保持数据一致，但是在操作的中间会临时地产生一些不一致的、其它软件不能安全解释或者操作的状态的时候，就出问题了。软件需要的是有能力标记一个数据转换为原子的：外部视图看到的数据只能是经历了整个变化或者根本没有变化。Linux 提供了少量简单的原子操作：如果某个体系结构提供了一些特别漂亮的实现机制，这些原子操作可以采用针对该体系结构的特殊实现方式。如果要把复杂操作搞成原子性的，Linux 则使用锁。[6] 锁就是操作一块数据的凭证，只要访问数据的所有软件都相互合作，一次只有一个线程可以操作数据。任何竞争的访问者在试图获取锁的时候都得等待当前线程结束。

想要在数据上执行一个原子性的操作序列的代码段称为临界区。所以我们要做的就是让所有想访问竞争数据的线程这样做：

```
acquire_lock(contended_data_marker)
```

[6] 锁是更一般意义上的信号量的一种极为简单的特例——但是从 Linux 的最小化的定义更简单，一开始更容易理解。

```
/* do your staff in the critical region */
release_lock(contended_data_marker)
```

每次调用这个代码的时候，不管到底是怎样实现的，总是要有一些时间花在锁的获取和释放操作中。当真的遇到竞争时，到达获取点的线程发现锁已经被用而必须等待，另一个到达释放点的线程必须想办法传递出消息说现在可以继续了。

当已知竞争者运行于另外一个 CPU 上且临界区只有几条指令时，可以让暂时受挫的获取者在原地回旋等待一直看着条件解除。这称为回旋锁 *(spinlock)*，Linux 对 SMP 系统提供了这种锁。

当竞争者可能是同一个 CPU 上另外的线程时，不可以用回旋等待（在等待线程回旋等待期间，持有锁的线程无法通过重新调度得到执行机会，因而没有机会执行完毕以释放锁，这样我们都陷入了死锁）。所以你需要一个更加重量级的锁，当线程获取锁失败时标记自己当前不能执行并调用操作系统调度器。还有，`release_lock()` 例程必须负责安排告知其它等待锁的线程可以再试一下。中止和唤醒 Linux 线程的代码不依赖于 CPU。但是测试锁的状态并在通常的没有竞争的条件下设置锁的动作有赖于一条原子性的 test-and-set（测试-设置）操作，其 MIPS 实现用到了下述的技巧。

在竞争者可能处于中断环境的地方（最后从中断入口点被调用、并且借用了某些随机选择的被中断线程的环境来这样做），Linux 依赖于在临界区执行期间禁止相关中断的做法。既然很难只屏蔽一个中断，那常常意味着在临界区附近屏蔽全部的中断。如前面提到的，中断环境中调用的代码受到许多约束以避免可能带来的麻烦。事实上，人们本来就期望中断处理程序跟内核其余部分之间只有简单而固定的交互。

MIPS 体系结构带了什么来帮助实现简单的原子操作和锁呢？

14.2.1 MIPS 体系结构和原子操作

MIPS 拥有一对连锁加载/条件存储指令。你用它们来实现在一个变量上任意的读-改-写 RMW 操作序列（就用 **ll** 读和 **sc** 写就行）。该序列本身不是原子性的。但是该序列若不能保证确实是原子运行的，存储操作就什么也不做，**sc** 返回一个软件可以测试的值。如果测试表明 **sc** 失败，软件可以重试一下 RMW 序列直到最后成功。

这些指令主要是为多处理器系统发明的，用来取代保证原子性的 RMW 操作。对多处理器系统这是一个很好的选择，因为避免了系统范围内的原子事务的开销。在一个大型多处理器系统上，为了确保访问没有触及被保护的数据，这种原子事务可能会停止所有的访存操作。

MIPS 的实现相当简单。**ll** 为每个 CPU 设置一个连锁位并且（对于多处理器和硬件多线程系统）记录加载的地址，这样 CPU 可以监控对该地址的访

问。**sc** 成功的条件——执行存储操作并返回 1 ——就是连锁位依然置位。如果监测到该变量（可能）被一些无关软件更新，CPU 就必须对该位清零。在多处理器系统中，外部访问的监测是通过保持高速缓存一致性的侦听硬件逻辑实现的。在单 CPU 上，任何异常都会破坏连锁位。[7]

Linux 的 `atomic_inc(&mycount)` 用该指令实现对任意整型变量的原子递增：

```
atomic_inc:
    ll   v0, 0(a0)                #a0 has pointer to 'mycount'
    addu v0, 1
    sc   v0, 0(a0)
    beq  v0, zero, atomic_inc
    nop
    jr   ra
    nop
```

如果你从这个例程退出，结果就是给 `mycount` 加了 1，并且该操序列——保证——没有被任何的 SMP 合作 CPU、本地允许其它软件运行的中断或者运行于一个硬件多线程的机器上的另一个线程所干扰。如果连续几次被外部写或中断影响而让该例程等上一段时间是可能的。但是——因为在 **ll** 和 **sc** 之间的序列很短——要是连试三次还不成功的可能性是极小的。

从上面的内容你也许可以看出，**ll/sc** 测试不适合于其中加载和存储离得很远的复杂的操作。如果等得太久，所有的连锁都会打破。

14.2.2 Linux 回旋锁

回旋锁用一个原子的 test-and-set 操作实现最简单的、最廉价的加锁原语，在多个 CPU 之间进行保护。怎样用 **ll/sc** 构建一个高效的回旋锁的具体推演留给读者作为一个练习。

与原子操作本身不同的是，回旋锁在单处理器（即单线程单处理器）系统上基本没有用处。

历史上，Linux 内核在支持内核抢占之前就实现了 SMP 安全性。“内核抢占”出现在中断导致运行于核心态的线程被调度出去而运行另一个线程的时候。完全可以构建一个 Linux 系统，其中核心态线程运行到要么时间片用完、要么自愿休眠（等待某个事件）、要么试图返回用户态。其实 2.6 版本之前的标准 Linux 内核就是这样做的，而且依然比一个运行着后台任务的当代的 Windows 系统的响应时间短。但是如果能正确实现内核抢占，响应性会更好。

在通往 2.6 版本内核的道路上，George Anzinger 观察到几乎所有需要软件保护免于内核抢占的临界区也都需要免受 SMP 并发性影响的保护，所以这两个区域已经由回旋锁分开了。回旋锁是宏，所以对单处理系统可以禁止它（编译成

[7] 在大多数实现中，连锁位实际上被对从中断返回必不可少的 **eret** 指令清零。

空行）。若构建 2.6+ 内核时定义了 `CONFIG_PREEMPT` 配置选项，回旋锁还顺便获得了在获取和释放操作之间禁止抢占的副作用。

好了，基本上就这些了。“几乎所有”是个危险的用词。只是偶尔情况下，某段数据操作本质上是 SMP 安全的（最明显的例子就是在寻址各 CPU 特有数据时）所以并不需要回旋锁，但的确需要防止抢占，因为同一 CPU 上其它内核代码可能在访问同一数据。加入内核抢占后保持 2.6 内核的稳定仍然需要相当多耐心的代码梳理和更多耐心的调试。

14.3 系统调用时发生什么

系统调用“仅仅是”内核实现的供用户程序用的例程。有些系统调用只是返回内核知道而外面不知道的信息（比如当天的精确时间）。

但有两个原因导致情况要比上面的复杂。第一个与安全性有关——人们认为内核在面对有错的或者恶意的应用程序代码时应当仍然是健壮的。

第二个原因与稳定性有关。Linux 内核应当能够运行专门为它生成的程序，但是也应当能够运行以前为同样的或者兼容的体系结构生成的应用程序。系统调用一旦定义，要删除就得经过许多的争论、工作和等待才可以。

我们先回到安全性。既然系统调用运行于核心态，核心态的入口就必须受到控制。对于 MIPS 体系结构，这是通过 **syscall** 指令由软件触发的异常来实现的，进入内核异常处理程序时，CPU 的 **Cause(ExcCode)** 寄存器域有一个特殊的代码（8 即 “sys”）。最底层的异常处理程序将会根据该域的值决定下一步干什么，并切换到内核的系统调用处理程序。

那只是单个入口点：应用程序设置一个数字参数选择几百个系统调用功能中的哪一个。系统调用值依赖于体系结构：MIPS 上对某个功能的系统调用号可能不同于 x86 上对同一功能的系统调用号。

系统调用参数尽可能在寄存器中传递，这有利于避免用户空间和内核空间之间不必要的数据拷贝。在 32 位 MIPS 系统中：

- 系统调用号放进 **v0**。
- 参数传递按照“o32”ABI 要求。大多数时间这等于说，最多可有四个参数在寄存器 **a0–a3** 中传递；但确实有些生僻的情况比较特殊，这些放在 11.2.1 节讲。[8]

  虽然内核采用类似 o32 的方式定义系统调用中的参数传递，那并不是说用户空间的程序必须要用 o32。良好的编程实践作法要求用户空间对操作系统的系统调用总是通过 C 运行库或者等价的机制传递，这样只需要维护一

[8] 许多参数传递的最偏僻的情形都不影响内核，因为内核并不处理浮点值，也不通过传值调用来处理结构体参数或者返回结构体结果。

处的系统调用接口。库可以从任何 ABI 转换成类似 o32 的系统调用标准，在 64 位 MIPS Linux 上运行 32 位软件时就是这样做的。

- 系统调用的返回值通常位于 **v0**。但是 `pipe(2)` 系统调用返回一个由两个 32 位文件描述符构成的数组，用了 **v0** 和 **v1** ——也许有一天会由另外一个系统调用完成同样的功能。[9]

  一切有可能失败的系统调用在 **a3** 中返回一个状态（0 表示良好，非零表示出错）。

- 为了和调用约定保持一致，系统调用保持 o32 要求在函数调用时保存的那些寄存器的值不变。

保证内核对系统调用实现的安全性需要适当的偏执。数组下标、指针、缓冲区长度都必须进行检查。并非允许所有的应用程序为所欲为，系统调用的代码在必要时（大多数时候，以超级用户 root 运行的程序可以做任何事情）要检查调用者的“权能(capabilities)。”

运行系统调用的内核代码处于进程环境——内核代码最宽松的环境。系统调用代码可以执行任何在发生 I/O 事件时要求线程睡眠的事情，或者访问需要交换进内存的虚拟地址。

进行数据传输时，你要依赖于应用程序提供的指针，而该指针可能是无效值。如果我们用了无效指针，就会引发一个异常，有可能导致内核崩溃——那是不能接受的。

所以从用户空间拷贝输入数据或者向用户空间拷贝输出数据可以通过专门用于该用途的函数 `copy_to_user()/copy_from_user()` 安全完成。如果用户真的传递了一个坏地址，将会导致内核异常。内核维护一个执行危险任务时可信任的函数[10] 列表：`copy_to_user()` 和 `copy_from_user()` 位于表中。当异常处理程序看到异常重新开始地址位于这些函数之一时，就返回到一个精心构造的错误处理程序。应用程序就会被发送一个粗鲁严厉的信号，但是内核安然无恙。

从系统调用返回是通过 **syscall** 异常处理程序结尾的 **eret** 指令返回的。重要的一点是切换回用户态和返回用户指令空间要同时执行。

14.4 MIPS/Linux 系统的地址转换

在我们开始了解怎样实现之前，先对整个工作有个整体轮廓。

图 14.1 是一个 32 位 MIPS/Linux 系统上的线程的存储器映射示意图。[11] 这个映射必须与硬件映射相一致，所以用户可访问的存储区必然位于下半部。

应当记住下面几点：

[9] 我听到内核维护人员对此异口同声地表示“我们誓死反对（除非踏过我们的尸体）。”

[10] 其实是一个指令地址范围的列表。

[11] 64 位 MIPS/Linux 系统的映射基于同样的原理，但是由于硬件映射更为复杂——参见图 2.2——因而相对麻烦一点。

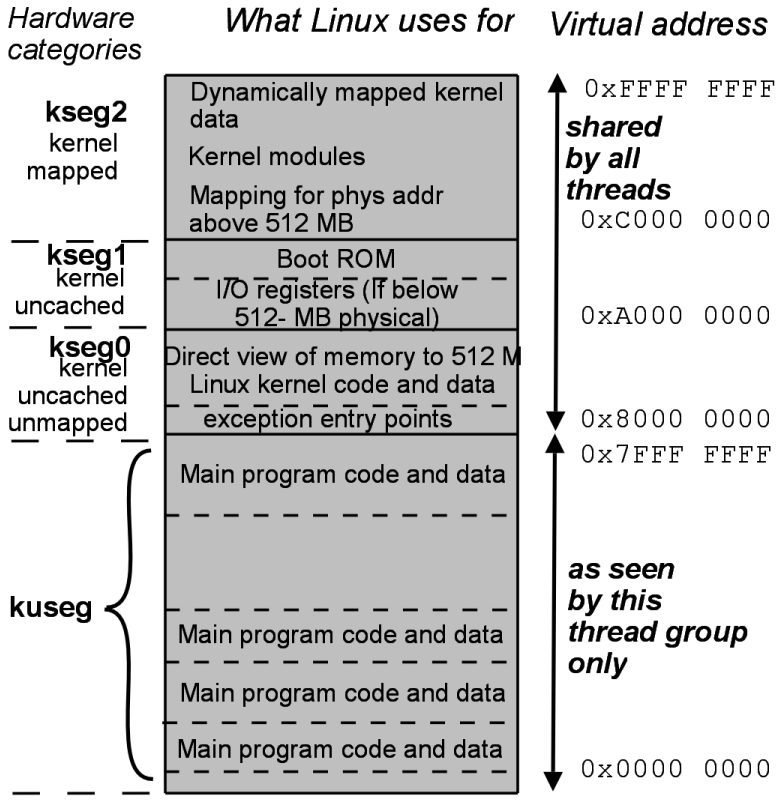

图 14.1: Linux 线程存储器映射

- 内核从什么地方开始运行：MIPS Linux 内核的代码构建为在 kseg0 区运行；虚拟地址从 `0x8000 0000` 向上。这个范围的地址仅仅是一个到物理内存低 512 MB 的窗口，无需 TLB 管理。

- 异常入口点：目前为止的大多数 MIPS CPU 中，这都由硬件固化到 kseg0 底部附近。最新的 CPU 可以提供 **EBase** 寄存器，对异常入口重新定位（参见第 3.3.8 节），主要是让多个共享内存的 CPU 能用不同的异常处理程序而不用费力去做特殊的存储器译码。在 Linux 内核中，就算有多个 CPU 也都运行同一个异常处理代码，所以这个特性在 Linux 中不大可能用到。

- 用户程序从什么地方开始运行：MIPS Linux 应用程序（运行于低特权级的“用户态”）虚拟地址从 `0` 到 `0x7FFF FFFF`。该区的地址在用户态可以访问，要经过 TLB 地址转换。

  应用程序的主程序构建时自接近零的地址开始运行。不会真为零——从虚拟零地址开始的一两页不做地址映射，这样企图使用空指针就会被当作内存管理错误捕获。应用程序的库函数部分，在加载或者更晚的时候递增加

载到用户空间。可以这样做是因为库函数构建为位置无关类型（参见第 16 章），能根据实际被加载的地址空间自动调整。

- 用户堆和栈：应用程序的栈初始设置到用户可以访问的空间（约 2 G 虚拟空间）的顶部并且向下增长。操作系统监测到对已分配的最低栈空间附近未映射的存储器访问时，会自动映射更多的页以满足栈的增长。

  同时，新的共享库或者直接用 `malloc()` 分配的用户数据及其后代从用户空间底部向上增长。只要这些空间的总和不超过 2 GB，什么事都没有：除了最大型的服务器以外，这个限制基本不成问题。

- 512 MB 以内的存储器：可以通过 kseg0 经过高速缓存访问或者通过 kseg1 不用高速缓存访问。历史上，Linux 内核假定自己可以直接访问机器的全部物理内存。对于用 512 MB 或者更少物理内存范围的小型 MIPS 系统，这是对的；在这种情况下，全部内存都可以在 kseg0（用高速缓存）和 kseg1（不用高速缓存）区访问。

- 512 MB 以上的“高位存储器”：现在 512 MB 即使对于嵌入式系统也已经不够了。Linux 有一个独立于硬件体系结构的“高位存储器”概念——要用特殊的、依赖于硬件体系结构的方式处理的物理存储器，对于 32 位 Linux/MIPS 系统，512 MB 以上的物理内存就是高位存储器。当我们要访问时，需要创建适当的地址转换数据项并且即时复制到 TLB。

早期的 MIPS CPU 追求在 Unix 工作站和服务器中应用，所以 MIPS 存储器管理硬件被设计成能够为 BSD Unix 提供存储器管理的最小硬件。BSD Unix 系统是第一个在 DEC VAX 小型机上提供真正的分页虚拟存储的 Unix 操作系统。VAX 在很多方面成为了此后统治计算机界的 32 位分页转换虚拟存储体系结构的典范；也许在 MIPS 里有一些 VAX 存储器管理组织的影响并不奇怪。但是这是一个 RISC，MIPS 的硬件做的要少得多。特别是，VAX（或者 x86）用微代码解决的很多问题在 MIPS 系统中都留给了软件去做。

在这一章，我们从 MIPS 硬件开始的地方开始，先考虑一个基本的 Unix 之类的操作系统及其虚拟存储系统的需求；但是这次我们的操作系统是 Linux。

我们将看到 MIPS 的硬件本质上是对这种需求的一种合理回应。至于真正的细节，请参阅第 6 章。

14.4.1 为什么要做存储器地址转换

存储器地址转换硬件（为了通用，我们称其为 MMU，即存储器管理单元 *memory management unit*）服务几个不同的目标：[12]

[12] 考虑到 MMU 对 Linux 的份量，在没有 MMU 的情况下可以构建一个大体可以工作的最小 Linux(uCLinux)，真让人有点不可思议——但那是另外一回事了。

- 隐藏和保护：用户特权级的程序只能访问程序地址位于 kuseg 存储区的（低程序地址）的数据。这样一个程序只能到达操作系统允许的内存区。

  此外，可以单独指定每个页面为可写或写保护；操作系统甚至可以停止一个意外覆盖自己代码的程序。

- 给程序分配连续的存储空间：有了 MMU，操作系统可以从物理上分散的页面构造连续的程序空间，允许我们从一个简单的固定大小页的缓冲池中分配存储器。

- 扩展地址范围：有些 CPU 不能直接访问它们全部的物理存储器范围。MIPS32 CPU 尽管是真正的 32 位体系结构，但对地址映射的安排使得非转换的的地址空间窗口 kseg0 和 kseg1（不依赖 MMU 表作地址转换）映射为物理存储器的低 512 MB。如果你需要更大范围的存储器映射，就必须经过 MMU。

- 使存储器映射适应你的程序：有了 MMU，你的程序可以使用适合自己的地址。在大的操作系统里，可能同一个程序有许多拷贝同时运行，让这些拷贝都用同样的程序地址要容易得多。

- 按需调页：程序运行时就好象需要的所有内存资源都已经分配了一样，但是操作系统实际上只在真正用到时才分配。访问未分配的地址区将产生一个异常，然后由操作系统处理；操作系统加载适当的数据到存储器后让程序继续运行。

  理论上（有时在计算机的教科书上）说，按需调页有用是因为这样允许你运行一个物理内存放不下的大程序。如果你有很多空闲时间的话，这话确实不错，但现实中如果一个程序真的需要用超过实际内存的空间，就会不断把自己的数据移出内存，以至于运行得非常非常慢。

  但是按需调页仍然十分有用，因为大程序充满了大量的至少在本次执行中不会运行的生僻代码。也许你的程序内置有对你今天用不到的数据格式的支持；也许程序有很多的错误处理代码，但是这些错误极少发生。在按需调页的系统中，没有用到的程序块从不需要读进内存。这样启动速度也快了，顺便讨好了缺乏耐心的用户。

- 重定位：程序入口点和预先声明的数据的地址在程序编译生成时是固定的——当然这一条对于那些用于 Linux 的全部共享库和大多数应用程序要用到的位置无关代码来说就不成立了。MMU 允许程序在物理内存的任意地址上都能运行。

Linux 存储器管理程序的工作的实质就是给每个程序都提供自己的存储空间。

好了，Linux 的几个概念是分开的：线程是调度单元，而地址空间（存储器映射）是保护单元。一个线程组中的许多线程可以共享一个地址空间。存储器转换系统感兴趣的显然是地址空间而不是线程。Linux 的实现是所有的线程都平等，所有的线程都拥有内存管理的数据结构。运行于同一个地址空间的线程，实际上共享这些数据结构的大部分。

但是就现在而言，如果我们只考虑每个线程一个地址空间的情形就比较简单，我们可以用老的概念“进程”同时指二者。

如果存储器管理工作不出错，每个进程的命运就独立于其它进程（操作系统也保护自己）：一个进程可能崩溃或行为不端但不会影响整个系统。对于大学里各系的运行学生程序的计算机，这显然是一个很有用的性质，但是即使最严格的商业环境也需要在支持经过测试检验过的软件同时，也支持试验性的或者原型软件。

MMU 不光是为了大型、完全的虚拟存储系统；即使小的嵌入式系统也能从重定位和更高效的内存分配中受益。任何想要在其中不同时间运行不同程序的系统都会发现，如果可以把程序的地址概念映射到随便一个可用的物理地址空间，一切都要容易得多。

多任务和不同任务地址空间的分离过去只用于大型计算机，后来迁移到个人计算机和小型服务器上，如今在消费类电子设备中的小的计算机上也日渐普及了。

然而，很少的非 Linux 嵌入式操作系统使用独立的地址空间。这可能不是因为该特性没有用处，而是由于嵌入式 CPU 及其可用的操作系统缺乏一致的特性。也可能因为一旦给系统加上独立的地址空间，就太接近于重新发明 Linux 而失去意义！

这对 MIPS 体系结构来说是一个意外的好事。1986 年为了简化工作站 CPU 而不得不进行的简约设计对于 21 世纪初期的嵌入式系统是个巨大的财富。即使小的应用程序，也被快速膨胀的代码所困扰，需要用尽所有已知的技巧管理软件复杂性；以 MIPS 为先驱的基于软件的灵活的方法极其有可能提供需要的全部东西。几年前，很难说服定位于嵌入式市场的 CPU 厂商加上 MMU；现在（2006 年）Linux 已经无处不在了。

14.4.2 基本的进程布局和保护

从操作系统的角度来看，低端存储空间(kuseg) 是个“沙盒”，在其中用户程序可以随便玩。如果由于程序失控丢失了自己全部的数据，这不关别人的事。

从应用程序的角度来看，该区可以自由使用来构建任意复杂的私有数据结构以完成工作。

在用户区内程序的沙盒里面，操作系统按需要（随着栈向下增长而隐含）即时提供更多的栈空间。同时也提供一个系统调用获得更多的可用空间，从预先声明的数据的最高地址开始向上增长——系统人员称之为堆。堆可供 `malloc()`

之类向程序提供额外内存空间的库函数使用。

堆和栈的空间供应的单位小到对于系统存储器适度节俭，大到能避免过多的系统调用或者异常。但是在每个系统调用或异常发生时，操作系统都会得到一个机会来控制应用程序的内存消耗。操作系统可以实施限制以保证应用程序不会因为占用内存太多而威胁其它关键的活动。

Linux 线程在操作系统核心保持自己的身份，当线程在内核里运行于进程环境（比如系统调用）时，其实只是调用一个精心编写的例程。

操作系统自己的代码和数据当然不能让用户空间程序访问。在一些系统上，这是通过把系统放到一个完全独立的地址空间做到的；在 MIPS 上，操作系统共享同一个地址空间，当 CPU 运行在用户程序特权级时，对这些地址的访问是非法的，会触发一个异常。

注意到虽然每个进程的用户空间映射到自己私有的真实存储，但特权空间是共享的。所有进程看到的操作系统代码和数据都位于同一个地址——操作系统内核是一个多线程但里头只有单一地址空间的系统——但是每个进程的用户空间地址访问自己单独的空间。可以信任运行于进程环境的内核例程会安全地协调操作，但是根本不必信任应用程序。

我们提到栈被初始化到可允许的最高用户地址：那样做可以支持使用大量存储器的程序。其结果导致使用的地址跨度很大（中间有一个巨大的空洞），这是这种地址映射的一个特征，在进行转换时必须要考虑。

Linux 把应用程序的代码映射为对本程序只读，这样代码可以被不同地址空间的线程安全共享——毕竟，许多进程运行同一个程序的情况很常见。

许多系统不仅共享整个应用程序，而且共享通过库调用访问的应用程序块（共享库）。Linux 用共享库做到这点，我们后面再接着讲那个故事。

14.4.3 进程地址到真实存储器的映射

支持这个模型需要哪些机制？

我们想要能够运行同一个程序的多个拷贝，各个程序拷贝最好使用不同的数据拷贝。这样在程序执行过程中，当程序加载时应用程序地址根据操作系统预先固定的方案映射到物理地址。

当然让操作系统在我们从一个进程环境切换到另一个的时候急急忙忙到处去拼凑地址转换信息是可能的，但这样会非常低效。取而代之的是，我们给每个活动线程的存储器映射一个 8 位的数，称为地址空间 *ID* 或 ASID。当硬件发现一个地址转换项的时候，寻找一个匹配当前 ASID 及地址的转换项，[13] 这样硬件可以容纳不同空间的转换而不会搞混。现在当调度不同地址空间的线程时软件要做的全部就是加载一个新的 ASID 到 **EntryHi(ASID)**。

映射机制也允许操作系统区分用户空间的不同部分。应用程序空间的有些部分（通常是代码部分）可以映射为只读；有些部分留下不做映射，在访问时自

[13] 实际上，我们可以看到有些转换项可以与 ASID 无关，即“全局”的。

陷，这意味着失去控制的程序可能会早点被停止。

进程地址空间的内核部分由所有进程共享。内核构建时就是为了在特定的地址运行，所以基本上不需要灵活的映射机制，可以利用 kseg0 区，把存储器映射资源留给应用程序使用。有些动态生成的内核区域是从映射的存储器（例如，用于容纳设备驱动程序的内核可加载模块等等都是映射的）方便地构建起来的。

14.4.4 分页映射

对地址映射曾经尝试过许多古怪的方案。直到 1980 年代中期前后，工业界还在想肯定有比固定大小的分页更好的解决方案。毕竟固定大小的页没有考虑应用程序的行为或需求，而且没有哪种自顶向下的分析会发明这个方法。

但是如果程序想要什么（当时想要的存储块大小），硬件就给什么，可用存储器很快就变成尺寸尴尬的碎片。当我们最后和外界接触时，他们的独轮车用的是圆形的轮子，而他们计算机可能用固定大小的页。

后来所有实际的系统都以页 ——固定大小的存储块——为单位映射内存。页的大小总是 2 的幂。太小的页可能需要太多的管理（大的转换表、一个程序需要太多的转换）；太大的页可能效率不搞，因为很多小数据也要占一个整页。最后 4 K 大小的页成为 Linux 上占据绝对优势的折衷结果。[14]

采用 4 KB 大小的页，程序/虚拟地址可以这样简单划分：

nn　　nn	11　　0
Virtual page adddress(VPN)	Adddress within page

页内地址位不需要做转换，所以存储器管理硬件只需要把地址的高位（传统上称为虚拟页号，即 VPN(virtual page number)）转换成物理地址的高位（物理帧号，即 PFN(physical frame number) ——没人记得当初为什么没叫做 PPN）。

14.4.5 我们真正想要什么

映射机制必须允许程序使用自己的进程/地址空间的具体地址，并把它高效地转换成为真正的物理地址来访问存储器。

一种好办法是弄一个表（即页表）为整个虚拟地址空间的每一页都保留一项，该项包含正确的物理地址。这显然是个相当大的数据结构，而且要保存到主存储器里面。但是有两个大问题。

第一个就是我们现在每次 load/store 需要访问两次存储器，这显然是个性能杀手。你可能预见到了这个问题的答案：我们可以用一个高速缓冲存储器来存储页地址转换数据项，只有在未命中这个高速缓存时才查找驻留内存的表。既然每个项代表 4 KB 存储空间，我们可以只用一个适度小的高速缓存就可以得到

[14]MIPS 硬件对于更大的页也没问题，16 KB 是下一个支持的页大小，对于许多现代的系统来说可能是一个更好的折衷。但是 4 KB 的页极为普及，其影响延伸到许多程序的构建方式；所以我们仍然基本上都用 4 KB。

一个满意的高命中率。(在这种方案发明的时候，存储器高速缓存还不多见，有时也叫做“旁视缓冲器(lookaside buffer)，”所以存储器转换高速缓存就成了转换旁视缓冲器，即 TLB；这个缩写名称一直流传了下来。)

第二个问题是页表的大小；对于 32 位应用程序，地址空间分为 4 KB 的页，有一百万个数据表项，至少要用 4 MB 空间。我们真得想办法让表小一点，否则就剩不下运行程序的空间了。

我们把解决这个问题的讨论推到后面，只是提一下很少真有程序用到 32 位寻址的 4GB 空间。多数中等的程序在它们自己的程序地址空间内部有巨大的空洞，如果我们能够发明一种方案，避免存储那些对应于空洞的“什么也没有”的转换表项，那么事情可能要好多了。

我们现在已经快要到 DEC 为其 VAX 小型机设计的存储器转换系统了，它对随后的体系结构有着极大的影响。概要如图 14.2 所示。

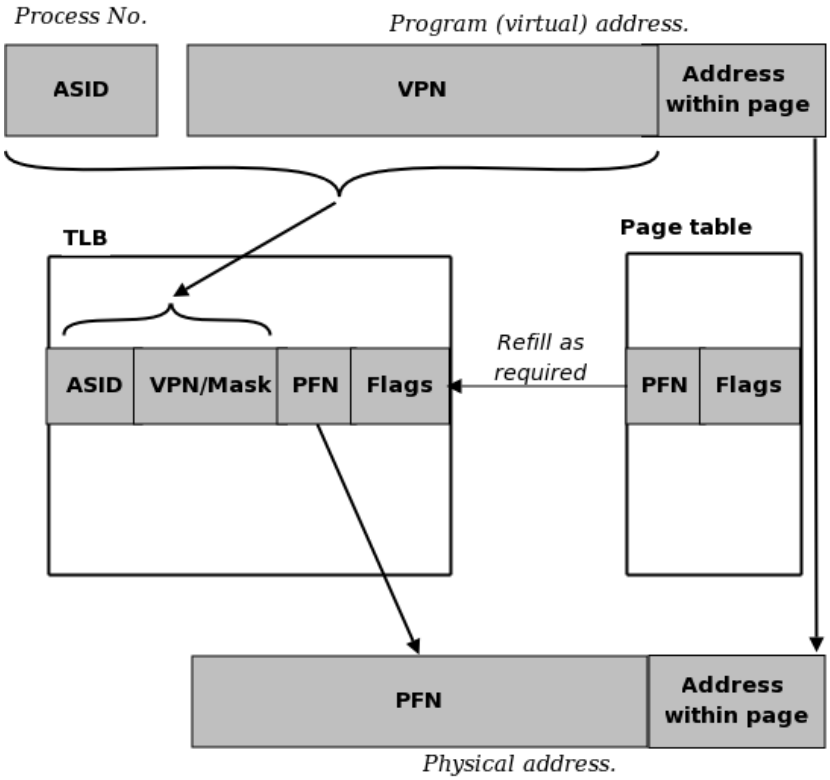

图 14.2: 想要的存储器地址转换系统

硬件工作的步骤如下：

- 虚拟地址分为两部分，最低有效位部分（通常是低 12 位）不做转换直接传递——这样转换总是以页（通常 4 K）为单位进行。
- 较高有效位，即 VPN，与当前线程的 ASID 拼接在一起形成一个唯一的页

地址。

- 我们在 TLB（地址转换的高速缓存器）看是否有该页的转换项。如果有，就给出物理地址的高位，我们就得到了要用的地址。

  TLB 是个专用的存储器，可以用各种方法有效匹配地址。可能有一个全局标志位告诉它忽略某些项的 ASID，这些 TLB 项可以用来映射某个范围的虚拟地址为每个线程所公用。

  类似的，VPN 可以和导致部分 VPN 地址不参与匹配的一些屏蔽位保存在一起，允许 TLB 项映射更大范围的虚拟地址。

  这两个特性在 MIPS MMU 中都有（但在一些很老的 MIPS CPU 中没有可变大小的页）。

- 通常有些额外的（标志）位与 PFN 放在一起用来控制访问权限——最明显的就是，只允许读不允许写。我们将在下一节讨论 MIPS 体系结构的标志位。

- 如果 TLB 没有匹配的项，系统必须（用驻留主存的页表信息）找出或者创建适当的页表项，加载进 TLB 然后再次运行转换过程。

  在 VAX 小型机里，这个过程由微代码控制，在程序员看来完全是自动的。如果你在存储器中构建了正确格式的页表并让硬件指向它，所有的存储器转换就能工作。

14.4.6 MIPS 设计的起源

MIPS 设计者想要找出一种方法用尽量少的硬件提供和 VAX 同样的功能。微代码控制的 TLB 填充是不可接受的，所以他们走出了大胆的一步，把这部分工作交给软件完成。

那就意味着除了有个寄存器存储当前 ASID 之外，MMU 硬件只是一个简单的高速、固定大小的转换表而已。系统软件可以把（通常也的确用）MMU 硬件看作驻留内存的整个页表的数据的一个高速缓存，所以把硬件表称为 TLB 是有意义的。但是 TLB 硬件中没有作为高速缓存的硬件，除了一点例外：当给出一个无法转换的地址时，TLB 触发一个特殊的异常（TLB 重填）来调用软件例程。为了帮助软件提高效率，TLB 设计、相关的控制寄存器以及重填异常的细节都要精心考虑。

MIPS TLB 总是在片上实现的。即使对于高速缓存引用，存储器地址转换步骤也是必须的，所以基本上处于机器的关键路径上。这意味着 TLB 容量必然小，尤其是在早期，所以必须靠设计的巧妙来弥补其容量的不足。

TLB 基本上就是一个完全的相联存储器。相联存储器的每一项都由一个关键字域和数据域构成；你给出关键字，硬件从该关键字匹配的项返回数据。相联

存储器确实非常好，但是其硬件成本很高。MIPS TLB 有 32 到 64 个数据项；这种容量的存储在半导体设计中不会太复杂。

当代所有的 CPU 都用这样一种 TLB，其中每个数据项都加倍，可以映射两个连续的 VPN 到两个互相独立的物理地址。这样一对数据项使得 TLB 只需增加少量逻辑就能让映射的存储器空间加倍，而不需要考虑大幅度的修改 TLB 的管理。

后面会看到，把 TLB 说成是全相联的；这种说法强调了 TLB 所有的关键字都真正并行地与输入值进行比较。[15]

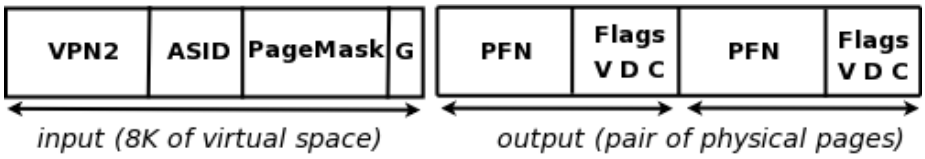

图 14.3: TLB 项的各个域

图 14.3 给出了 TLB 数据项的示意图。就现在而言，我们假定页大小为 4 KB。TLB 的关键字——输入值——由三个域组成：

- VPN2: 页号就是虚拟地址的高序位——当你去掉页内地址 12 位后剩下的部分。VPN2 中的“2”强调每个虚拟项有双倍的输出域从而映射 8 KB 地址。虚拟地址的位 12 选择一对物理地址项中的第一个或第二个。

- PageMask: 控制虚拟地址的多少部分和 VPN 相比较，多少部分传递到物理地址；减少匹配位可以映射更大的存储区。“1” 位表示忽略相应的地址位。有些 MIPS CPU 可以设置成每项最多映射 16 MB。被忽略位中的最高有效位用来选择奇偶项。

- ASID: 标记该项地址转换属于特定的地址空间，所以只能在给出地址的线程的 **EntryHi(ASID)** 设置等于该值时才参与匹配。

  G 位如果为 1 则关闭 ASID 匹配，使得该转换项适用于所有的地址空间（从而这部分地址映射在所有地址空间之间共享）。ASID 有 8 位：熟悉操作系统的读者可能会意识到即使是 256 作为大型 Unix 系统上同时活动的进程数的上限也还是太小了。但是，只要给这个环境下的“活动的”赋予特殊的解释“可能在 TLB 中有地址转换项，” 256 还算是一个合理的上限。操作系统软件必要时不得不回收 ASID，这将涉及到在 TLB 中对那些从活动状态降级的进程的地址转换项进行大清洗。这可是件吃力不讨好的活，但是吃力不讨好的活操作系统不得不干的多了；256 项应该足以保证 TLB 清洗的次数不会多到造成性能问题。

[15]把通用的具有 32 项、每项包含一对虚拟地址的 TLB 说成是一个 32 路组相联、每组两个数据项的高速缓存，虽然显得有点学究气，但确实没错。

从编程的角度讲，最简单的是 **G** 位保存在内核的页表中和输出域放在一起。但是当你做地址转换时，**G** 位又属于输入。在 MIPS32/64 CPU 里，两个输出端值进行“与”操作后生成要用的值，但是要得到有意义的结果，你必须保证两半的 **G** 位设置一样。

TLB 的输出端给出物理页号和少量但是够用的标志位：

- 物理页帧号(PFN): 这是切除低位（如果代表 4 KB 的页就是低 12 位）后的物理地址。
- 写控制位(D): 设为 1 允许对该页写入。“D”是因为该位被称为“dirty” 位；原因请参阅下节内容。
- 有效位(V): 若该位为零则该项不可用。这看上去好像没有什么意义。既然不能用，为什么要把这种记录加载进 TLB 呢？这有两个原因。第一个是因为每一项转换一对虚拟地址，可能只要其中一个。另一个是重填 TLB 的软件为了优化速度，而不想检查特例。如果程序在能够使用驻留内存的页表所指向的页之前还需要进一步的处理，可以保留驻留内存的页表项而标记其为无效。在 TLB 重填之后，这会导致另一种自陷来调用特殊的处理，而不需要每次软件重填的时候都作检测。
- 高速缓存控制位(C): 这个 3 位的域的首要目的是区分可作高速缓存的（cachable）和不作高速缓存的（uncached）区域。

  但是那还剩下六个取值，用于两个有点互不相容的目的：在共享存储器的多处理器系统中，不同的值用来给出存储器是否共享的提示（此时硬件需要尽力保证整个机器上各个高速缓存的数据一致性）。在“嵌入式”CPU 中，不同的值选择不同的局部高速缓存管理策略：比如透写还是回写等等。具体请参阅你的 CPU 手册。

地址转换现在简单了，我们可以把上面的描述详细展开如下：

- CPU 生成一个程序地址：这可能是取指或数据读写操作——而且其程序地址并不位于 MIPS 地址空间特殊的非映射区。

  低 13 位是分开的，剩下的 VPN2 和当前 ASID（**EntryHi(ASID)**）合在一起，在 TLB 中进行查找。查找匹配还受到各个 TLB 项的 **PageMask** 和 **G** 域的影响。
- TLB 匹配关键字：如果没有相匹配的关键字，就触发一个 TLB 重填异常。但是如果有匹配，就选择该项。虚拟地址的位 12 用来选择要用哪半面的物理地址。

  来自 TLB 的 PFN 粘到程序地址的低位上构成一个完整的物理地址。

- 地址有效吗？要察看 V 和 D 位。如果不有效，或者 D 位是零而试图存储，CPU 就会触发异常。与所有地址转换自陷一样，**BadVAddr** 寄存器会被填上相应的程序地址；与所有的 TLB 异常一样，TLB 的 **EntryHi** 寄存器会预加载相应地址的 VPN。

  便利寄存器 **Context**（64 位 CPU 则为 **XContext**）的 **BadVPN2** 域将会（部分）预加载我们在 TLB 异常重填期间未能转换的虚拟地址的若干位。但是 MIPS32/64 规范对于这些地址域在别的异常中的行为没有明确说明。在其它异常中坚持只用 **BadVAddr** 可能是个好办法。

- 是否高速缓存？如果 C 位置位，CPU 在高速缓存中查找物理位置数据的拷贝；如果没有找到，就从存储器中取出数据，并在高速缓存中留下一个拷贝。当 C 位清零时，CPU 既不查找也不填充高速缓存。

当然了，TLB 中数据项的个数允许你只转换相对少量的程序地址——几百 KB。这对于大多数系统来说远远不够。TLB 几乎总是用作一个由软件维护的高速缓存，缓存的内容就是位于内存的比其自身容量大得多的地址转换数据表。

当程序地址在 TLB 中查找失败时，就发生一次 *TLB 重填自陷*。[16] 系统软件要完成如下的工作：

- 先看看有没有正确的地址转换数据；如果没有，该自陷将调用处理地址错误的软件。
- 如果有正确的地址转换，系统将构造一个实现该转换的 TLB 项。
- 如果 TLB 已经满了（实际运行的系统中几乎永远是满的），由软件选择一项可以丢弃的。
- 软件将新的地址转换项写入 TLB。

参见第 14.4.8 节以了解在 Linux 中的具体做法。

14.4.7 跟踪被修改的页（模拟“Dirty”位）

给应用程序提供存储器页使用的操作系统常常想要跟踪自上次操作系统（从磁盘或者网络）取得或者保存该页的拷贝以来该页是否作了修改。未修改的页（即“净(clean)”页）可以直接丢弃，因为再次要用时可以很容易从文件系统恢复。

用操作系统的行话来说，修改过的页叫做“脏(dirty)”页，操作系统必须仔细处理，一直到应用程序结束或者脏页因为保存到后备存储器而“净化。” 为了辅助这种操作，CISC CPU 通用的做法是在驻留内存的页表中维护一个位，用以

[16] 这究竟应当称作是“”TLB 失效(miss)”（即刚发生的）还是“TLB 重填(refill)”（即我们将要做的）？恐怕我们可能在 MIPS 文档中二者都用。

表示发生了对该页的写操作。MIPS CPU 即使在 TLB 数据中也不支持这个特性。页表中的 D 位（可从 **EntryHi** 寄存器找到）是写允许位，当然是用来标记只读页的。

所以采用了如下的技巧：

- 当可写的页首次加载进内存时，将其页表项的 D 位清零（表示只读）。
- 当试图对该页写入时，就会发生自陷；系统软件将认出这是一个合法的写操作但利用这个事件在驻留内存的页表中设置一个“modified(修改过)”位——因为该位处于 **EntryLo(D)** 的位置，使得将来的写操作可以不发生异常而正常进行。
- 你也要设置 TLB 项中的 D 位，这样写操作可以进行（但是鉴于 TLB 项的替换是随机不可预测的，以此作为记住被修改状态的做法是没有用的）。

14.4.8 内核对 TLB 重填异常的服务过程

MIPS 的 TLB 重填异常总是有自己独自的入口点（至少在 CPU 尚未处于异常模式时如此；在 Linux 中处于异常模式时发生 TLB 重填异常是一个致命错，因此不予考虑）。

异常例程被调用时，硬件已把 **EntryHi(VPN2)** 设置为刚才未能转换的地址的页号：**EntryHi** 设置成恰好就是创建相应地址映射的新 TLB 项所需要的格式。

硬件也要设置一些其它地址相关的域，但是这些我们一律都不用。

特别的，我们不用第 6.2.4 节介绍的用 **Context** 找出相关的页表项的方便易用的“MIPS 标准”做法。MIPS 标准做法要求在 kseg2 构建一个（名义上很长的）线性页表（实际上不会真的占用太多空间，因为 kseg2 区的地址要进行映射，表中大范围的空洞不会映射到真实的存储器）。但是这样就要求每个线程组有各自不同的内核映射，Linux 不想要这种结果。

代替的做法是，Linux 的 TLB 重填页表组织成一个三级页表（分别称为“全局级(global)”、“中间级(middle)”和“PTE”）。但是巧妙的使用 C 语言的宏可以不用改动代码而让中间级完全不出现——两级结构对于 32 位 MIPS 足够了。[17] 这样你可以沿着链接找到任何你想要的 TLB 项的数据——如果该数据存在的话，如图 14.4 所示。

这种结构对于内核存储器的使用相对比较节省：一个大点的 50 MB 虚拟地址空间的 Linux 程序大约有 12 K 个 4 KB 大小的页，每页需要四字节的 PTE：这需要 48 KB 或者 12 个页。这是一个保守的估计，因为地址空间内部会有空洞。但是合理大小的线程组的映射也就需要占用 15 页左右。内核（kseg2）映射占用的要多些，但是因为内核映射对所有存储器映射都通用，所以只有一组 PTE。

[17] 但是对于 64 位 MIPS 上扩展的虚拟存储器空间，所有三级都是必须的。

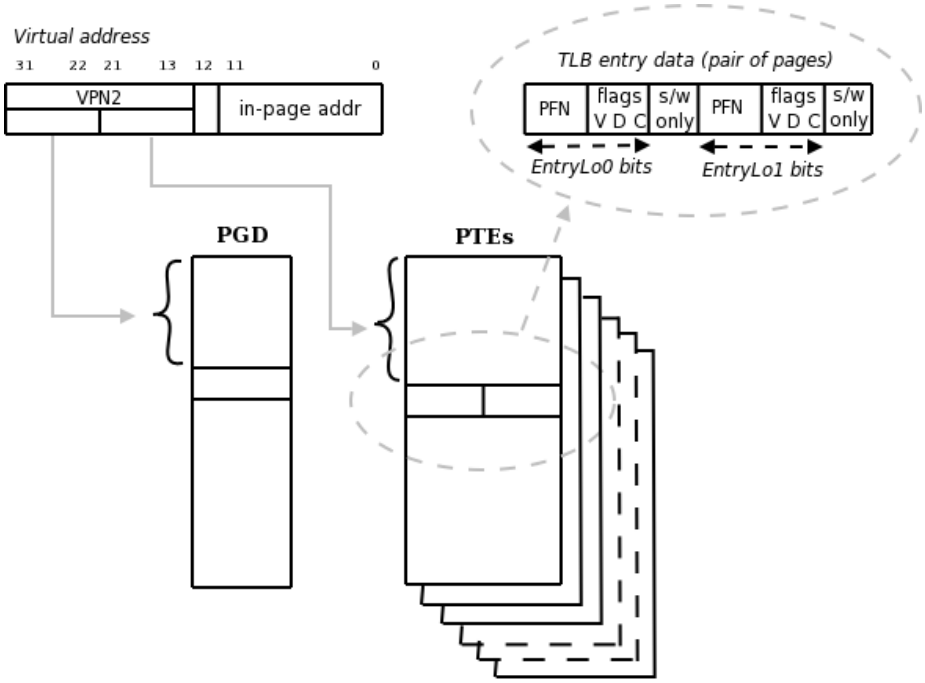

图 14.4: Linux 两级页表（32 位 MIPS 设计）

大多数 32 位 CPU 物理地址空间也限制为 32 位：此时，在 **EntryLo0–1** 寄存器中有六个高位未用。Linux 回收这些位用来记录关于页表项的一些软件状态。

这的确意味着对这 32 位的 TLB 重填处理程序需要做两次索引计算和三次访存。那当然比原始的 MIPS 方案（其中索引计算由 **Context** 寄存器完成的，唯一的读存就是从最后的页表读取）的少量指令序列要长，但是也还不错。

你可能注意到这里的叙述只是针对某类特定的 MIPS CPU。每个 Linux 系统的内核都不同吗？当然不是：但是在当前的 Linux/MIPS 的内核里，某些关键例程——包括 TLB 未命中异常处理程序——是由表驱动的内核软件在启动过程中生成的二进制代码，针对具体 CPU 作了裁减。

作为例子，下面给出为一个基于 32 位 MIPS 24-K 核的系统自动生成的 TLB 未命中处理程序。

```
tlb_refill:
    # (1) get base of PGD into k1
    lui     k1, %hi(pgd_current)

    # get miss virtual address (VA)
    mfc0    k0, c0_badbaddr # (2) moved up to save a cycle
    lw      k1, %lo(pgd_current) (k1)
```

```
srl     k0, k0, 24
sll     k0, k0, 2       # (3) shift and mask VA for PGD index
addu    k1, k1, k0      # got a pointer to the correct PGD entry

# get VA again, this time from Context register
mfc0    k0, c0_context  # (4) moved up to save a cycle
lw      k1, 0(k1)       # OK, read the PTE pointer

srl     k0, k0, 1       # (5) Context register designed for 2x64-bit,
                        # entry, ours are half that size
andi    k0, k0, 0xff8   # (6) mask out higher VPE entry

# load the TLB entries
lw      k0, 0(k1)       # (7) will lost a cycle here
lw      k1, 4(k1)

srl     k0, k0, 0x6     # (8) shift-out software-only bits
mtc0    k0, c0_entrylo0
srl     k1, k1, 0x6     # same for other half
mtc0    k1, c0_entrylo1

ehb                     # (9) wait for CP0 values to be ready
tlbwr                   # (10) write into TLB (somewhere)
eret                    # (11) back to user
```

TLB 重填代码的说明

(1) 在异常处理程序中，软件约定允许我们无偿使用两个寄存器（**k0–1**）。其它所有寄存器都还保留着应用程序的数据，我们不能碰。

(2,4,5) 该代码因为一个指令序列和另一个交替而搞得比较难读。这样做是为了效率：24-K CPU 读存需要基址寄存器提前一个周期准备好，所以如果你在上一条指令计算基址就会导致一个周期的停顿。在 (2) 和 (4) 处我们找到了一个没有依赖的指令进行交替，但在 (5) 处就没有。

(3) 从图 14.4 可见，我们用虚拟地址的高位和未命中的转换地址（位于 **BadVAddr**）作为 PGD 的索引。因为 PGD 在每一项中都有指针（四个字节），我们需要把索引值左移二位得到字节偏移量。你不能简单的右移 22 位，因为你可能生成一个没有字对齐的指针，这对 MIPS 的字加载操作是非法的。

(4) 参见 (2,4,5)。

(5) 只有两个寄存器可以用，我们无法保存原先的虚拟地址。所以现在再次获取其值，但是这次是从 **Context** 寄存器读取，其中有我们需要的“VPN2”域的低位。对于 64 位 CPU ——需要保持两个 64 位的项来填充一对 **EntryLo0–1** ——结果刚刚好，VPN2 上移 4 位生成一个由 16 字节项构成的表的索引。但是 MIPS32 上的 Linux/MIPS 的 PTE 表中每项是 8 个字节，所以我们还要右移一位。

(6) 因为不想要**Context** 的高位，所以我们将其屏蔽出去。即使在页号之上的位也并非总是为零：SMP Linux 系统用到仅由普通的读写位构成的 **Context(PTEBase)** 域，在里头存放一个 CPU ID。

(7) 参见 (2,4,5)。

(8) 在这类拥有 32 位物理地址空间的系统上，**EntryLo** 寄存器只有 26 个有意义的位。Linux 把其余 6 个用作软件标志，我们通过移位去掉了这几位。

(9) 这条执行遇险防护指令保证后续的指令延迟至对 CP0 的 **EntryLo0–1** 寄存器的写入操作生效之后。在许多 CPU 上，从 **mtc0** 指令向 CP0 寄存器的写入操作在流水线的后期才生效。

(10) **tlwr** 将一个完整的 TLB 项写入 TLB 的一个“随机”位置。事实上，它从 **Random** 获取索引，该寄存器在各个 TLB 位置之间连续循环计数，但是 TLB 未命中发生的时间足够随机，实际效果很好。

注意 TLB 的内容涉及到四个 CP0 寄存器：我们刚刚加载了 **EntryLo0–1**，**EntryHi** 由 TLB 未命中异常硬件自动设置，**PageMask** 由 Linux/MIPS 内核维护为对应于 4 KB 页的常数。

(11) **eret** 把我们带回到遭受 TLB 未命中的指令，这回做得应该好一点。注意 **eret** 还是一个 CP0 遇险防护指令，所以直到 **tlbwr** 完成之后才允许对返回用户的指令取指。

14.4.9 TLB 的维护以及注意事项

MIPS TLB 只是一些地址转换数据项的集合。每一项完全是由于重填处理程序的行动才进入 TLB。当然大多数项也以同样方式被覆盖。但是有时候内核想要改变某个页表项，这时作废 TLB 中的拷贝就很重要了。

对于单个的项，我们可以用虚拟地址+原始项的 ASID 查找 TLB，执行 **tlbp**: 如果那里有匹配项，我们可以用一个无效的转换覆盖该页的转换项。

然而，有时候你可能要大规模的动作。ASID 机制允许 TLB 容纳 256 个不同的存储器映射项，但是 Linux 系统在启动和关机之间通常运行的进程数要多于 256 个。所以有时候要清除一大批地址转换项，因为要回收 ASID 用于新进程，这样的老的地址转换就完全错了。

做到这一点没有简洁的方法。你只得依次遍历所有的 TLB 项（索引值在 **Index** 寄存器里面），把每一项读到常规寄存器中，检查 **EntryHi(ASID)** 与 ASID 的值，作废与要回收的 ASID 值相匹配的每一项。

14.4.10 存储器地址转换和 64 位指针

当 MIPS 体系结构发明的时候，32 位 CPU 已经出现了一段时间了，当时最大的程序的数据已经快增长到 100 MB ——地址空间只剩下 6 位左右。[18] 因此有足够的理由要精心使用 32 位空间，不要因为昂贵的分段而减少空间；这就是为什么（以用户特权运行的）应用程序为自己保留 31 位寻址空间。

当 MIPS III 指令集于 1991 年引进 64 位寄存器时，一度领先于整个业界，正如我们在 2.7 节讨论的，MIPS 可能比 32 位地址真正显出不足要早四到六年。寄存器宽度的加倍只要给地址空间增加几个位就可以保证未来够用了；更重要的是得考虑操作系统数据结构潜在的爆炸性增长而不是有效地利用全部地址空间。

由于基本的 64 位存储器映射对实际的地址空间的限制在一段时间内还不会达到；理论上允许可映射的用户和其它空间在不用重新组织的情况下就能一直增长到 61 位。但是到目前为止，40 位的用户虚拟空间已经很充足了。大多数其它的 64 位 Linux 系统采用 8 KB 的页，但是 8 KB 的页在 MIPS 中很麻烦。MIPS TLB 的单个转换项可以映射要么 4 KB 要么 16 KB 大小的页，但不能映射 8 KB 的页。这样 64 位 MIPS 内核用 4 KB 的页，把 16 KB 的页作为在勇敢的未来一个不错的选项。

如果你回头看一下图 14.4 再想象一下在 PGD 和 PTE 之间的一组中间（PMD）表，就能很轻松地在三级页表中解析 40 位虚拟地址。我们把细节留给那些愿意阅读源码的热心读者。

[18] 历史上，应用程序对存储器空间的需求看上去大约每年增长 $\frac{3}{4}$ 位。

第 15 章　Linux 内核中关于 MIPS 的专门问题

Linux 内核的大部分都是用可移植的 C 写的，大部分的代码不用进一步处理就可以直接移植到象 MIPS 这类清晰的体系结构上。在上一章我们看到了一些围绕着异常处理和存储器管理的机器相关的代码。本章看一下其它需要特殊的 MIPS 代码的地方。

前两节涉及大多数 MIPS CPU 在编程方便性和硬件简单性之间的权衡：首先，MIPS 高速缓存常常要用软件管理，其次，MIPS CP0（CPU 控制）操作有时需要直接考虑流水线效果。我们下来将很快看一遍有关 MIPS 的多处理器（SMP）Linux 系统需要了解的知识。最后一节瞧一眼怎样用传奇式的汇编语言代码加速一个使用频率很高的内核例程。

15.1　直接管理高速缓存

在 x86 CPU 中，也就是 Linux 出生长大的地方，高速缓存基本是不可见的，由硬件维护一切，就好象在直接和内存打交道一样。

MIPS 系统不是这样，其中许多 MIPS 核的高速缓存没有任何类型额外的“一致性”硬件。Linux 必须处理好几个方面的问题。

15.1.1　DMA 设备存取

DMA 控制器直接写（使得高速缓存内容过时）或者读（可能错过尚未写回的高速缓存数据）存储器。在一些系统上——特别是 x86 PC 上——DMA 控制器会设法把有关数据传输的信息告诉硬件高速缓存控制器，高速缓存控制器必要时自动作废或者回写高速缓存内容，使得整个过程是透明的，就象 CPU 在直接读写存储器一样。这样一个系统称为“I/O-高速缓存一致性(I/O-cache coherent)”或者简称“I/O 一致性(I/O coherent)。”

很少有 MIPS 系统是 I/O-cache 一致的。大多数清形下，DMA 传输在没有告知高速缓存逻辑的情况下就开始了，设备驱动程序必须管理高速缓存以保证不会使用高速缓存或存储器中的过时数据。

Linux 有个 DMA API 向设备驱动程序输出管理 DMA 数据流的例程（其中许多例程在 I/O-cache 一致的系统中为空）。你可以在随 Linux 内核源码提供的包括 `Documentation/DMA-API.txt` 在内的文档中看到这些例程。实际上，如果你在编写或移植设备驱动程序，就应看看这些文档。

当驱动程序请求分配缓冲区时，可以选择：

- “’一致性”存储器：由 Linux 保证“一致性”存储器与 I/O 之间的一致性，可能要付出一些性能代价。在 MIPS CPU 上最可能的实现就是不用高速缓存，这时付出的性能代价是不可忽略的。但是一致的缓冲区是处理复杂设备控制器的仅占用少量内存的控制数据结构的最佳方式。

- 使用非一致存储器作缓冲区：既然一致的存储区在许多 MIPS 系统上不使用高速缓存，这对于大的 DMA 缓冲区可能导致很差的性能。

  所以对于大多数常规的 DMA，该 API 提供了名字象 `dma_map_xx()` 的一类调用。它们提供适合于 DMA 的缓冲区，但是该缓冲区不是 I/O 一致的，除非系统能够以低成本实现通用的 I/O 一致性。

  内核存储器分配程序确保缓冲区位于 DMA 可以达到的存储区域，隔开不同的缓冲区使它们不会共享同一个高速缓存行，并以 DMA 控制器可以使用的形式提供一个地址。

  因为使用了非一致的缓冲区，就提供了一些操作缓冲区的调用，以及在 DMA 前后执行高速缓存作废或回写操作的调用：叫做 `dma_sync_xx()`，API 里还包括在什么时候怎样调用这些函数的说明。

  对于实现了真正的高速缓存一致性的硬件，“sync”函数为空。

  这一段 API 文档的用词有些不妥。有一个极少用到的 API 扩展，其函数名包含“nocoherent（非一致的）”这个词，但是你不应当用它，除非你的系统实在很古怪。一个常规的 MIPS 系统，即使不是 I/O 一致的，也能够而且应当让用了标准 API 的驱动程序正常工作。

按照操作系统的标准，这些都是非常简单直接的。但是许多驱动程序开发人员的工作都是在用硬件管理高速缓存的机器上做的，他们系统上的‘sync”函数只有一个空壳子。如果忘记了在适当的地方调用正确的 sync 函数，他们的软件用起来仍然没问题：直到你把这些软件移植到一台需要直接管理高速缓存的 MIPS 机器上时才会出问题。所以从其它地方拿来的驱动程序代码要小心。给未来的 CPU 加上高速缓存管理硬件，最有说服力的理由就是为了让软件移植工作更加顺利。

如果你有兴趣看一眼 Linux 的实现，了解一下怎样从 MIPS 指令出发构建高速缓存同步函数，4.6 节可以作为一个参考。

15.1.2 动态生成随后要执行的指令

为自己生成指令的程序生成的指令可能只是留在数据高速缓存里而还没有写入存储器中，也可能让该指令所在的指令高速缓存中的内容沦为了过时的数据。

这不是内核特有的问题：事实上，在一些加速语言解释器的“即时(just-in-time)”翻译器之类的应用程序中碰到的机会更多。讨论怎样以可移植的方式解决此类问题超出了本书的范围，但是 MIPS 上的任何做法将来都应该建立在 **synci** 的指令基础上。这是理想的情形：**synci** 只在 MIPS32/64 规范的 2003 年第二版中才定义，许多没有该指令的 CPU 仍然在用。在这种 CPU 上，一定有个特殊的系统调用来利用特权指令 **cache** 执行必要的 D-cache 回写和 I-cache 作废操作。**synci** 指令的细节可以在第 8.5.11 节可以找到。

15.1.3 高速缓存/存储器映射问题

虚拟高速缓存（真实存在的高速缓存，只是采用了虚拟索引和标签）好象是一次奇妙的的免费搭车，因为整个高速缓存查找过程可以早点开始并且和基于页的地址转换并行进行。

一旦存储器映射有变化，简单的虚拟高速缓存就必须清空，除非高速缓存很小否则这种代价难以忍受。如果你用 ASID 扩展虚拟地址，就可以区分不同进程的地址转换。

操作系统程序员知道为什么虚拟高速缓存是个馊主意：虚拟高速缓存的问题就是高速缓存中的数据生存期可以跨越页表的变化。一般而言，虚拟高速缓存在每个地址映射废止后都要重新检查。这样做成本很高，所以操作系统工程师想尽办法减少更新，因而漏掉了一些生僻少见的情形，结果就引入了软件缺陷。

在为了让 Linux 能够成功运行虚拟高速缓存的系统而进行的大胆的尝试中，作为向高速缓存有问题的体系结构进行移植的工作的一部分，内核提供了一系列的规则和函数调用。这些函数名字都以 `flush_cache_xx()` 打头，在内核文档 `Documentation/cachetlb.txt` 中有描述。我不喜欢用“flush”这个词描述高速缓存操作：这个词已经用来表示了太多的意思。要小心注意，在 Linux 内核中一次“cache flush”是指为了删除与过时的存储器映射有关的高速缓存项而执行的某些操作。在一个全部高速缓存都采用物理索引和物理标签的系统中，这些调用什么都不用做。

幸运的是，虚拟 D-cache 在 MIPS CPU 上很少见。有些新近的 CPU 带有虚拟的 I-cache: 实现了文档中描述的“flush”功能，那就应该没问题。但是采用物理标签而用虚拟索引的 L1 高速缓存在 MIPS CPU 上很常见。它们解决了本节所述的问题，但是导致了另外一个称为“高速缓存重影” 的问题——请接着阅读下一节。

15.1.4 高速缓存重影

我们现在碰到的东西危害性更大。MIPS CPU 设计者第一批认识到可以把高速缓存采用虚拟地址作为索引的好处与采用物理地址作为标签的好处结合到一起。这样可能导致高速缓存重影——详细解释请参阅第 4.12 节。

R4000 CPU 是第一个采用虚拟索引高速缓存的。按照原先的设想，CPU 总是带有 L2 高速缓存的（L2 高速缓存的存储器不在片上，但是高速缓存控制器集成到了 CPU 里），用 L2 高速缓存检测 L1 高速缓存重影。如果你加载的一行数据与已经位于 L1 中的一行发生重影，CPU 就会产生异常，可以用这个异常来执行清理工作。

但是通过略去 L2 高速缓存芯片以及与它连接的引脚而生产一种占用面积更小、成本更低的 R4000 变体的诱惑力太强大了。当代的 UNIX 系统曾经以一种固定不变的方式使用虚拟存储器，这就意味着可以通过控制存储器分配来避免重影。现在回过头来看，我们认识到产生重影属于硬件的缺陷，通过仔细的存储器管理来避免重影只是为了克服这个缺陷。但是这种做法有效，以至于人们后来忘记了这点，最后这个缺陷演变成了一个特色。

基本上有两种处理高速缓存重影的方法。

第一种是试图保证当某个页共享时，所有引用该页的虚拟地址都具有同样的“页色”（就是说，页的虚拟地址可以不同，但是它们之间相差必须是高速缓存每组容量的整数倍）。在“同色”页中两次可见的任意数据都会存储在高速缓存同一索引位置而得到正确处理。保证一个页在所有用户空间的映射都属于同色是可能的——更多内容参见第 10.3.4 节。

但是与老式的 BSD 系统不同，Linux 提供的一些特性使得保证正确的页色变得不可能。这包括那些同一页既有用户空间映射又有内核空间映射的情形（在 MIPS 内核上，许多情况下内核“映射”是一个 kseg0 地址）。所以 MIPS 移植需要有特殊的代码检测这种情形，并清理掉任何老的重影映射。

高速缓存/TLB 文档（即上节提到的 `Documentation/cachetlb.txt`）做了一个勇敢的尝试，试图把高速缓存重影当成一般的虚拟高速缓存的“另一种症状”处理。它提供了一些注意事项，主要是怎样配置内核来尽量处理页色，怎样处理内核/用户空间的重影。

围绕高速缓存重影的修正工作永远不会结束。过去的这些年，人们期望 Linux 提供更多激动人心的功能。程序员苦苦挣扎着修正重影可能破坏合法代码的地方，但是随着新功能加入到操作系统，又会出新的问题。我想 CPU 设计者现在认识到了依靠软件克服重影会造成许多维护问题：越来越多的新 CPU 至少 D-cache 没有重影问题。

MIPS 曾经是一个很有影响力的体系结构，所以其竞争者也忠实地照抄了它的错误；[1] 结果呢，其它体系结构里也有了高速缓存重影。

[1] 如果模仿是最真诚的奉承形式的话，那么模仿一个体系结构的错误肯定相当于偶像崇拜。

15.2 CP0 流水线遇险

如我们在上面的例子中所看到的，依赖于某些 CP0 寄存器值的 CP0 操作需要仔细处理：流水线对于这些操作系统专用的操作不总是隐藏的。操作系统当然是所有 CP0 操作发生的地方。

在一个现代的 CPU（符合 MIPS32/64 第二版规范的）上，可以用第 3.4 节的遇险防护指令（**eret**、**jalr.hb** 和 **jr.hb**）。

在老一点的 CPU 上，只有异常入口和 **eret** 指令能保证提供 CP0 遇险防护。所以当你写一段依赖于 CP0 指令的正确完成的代码时，你可能需要加上经过计算出来的一定数量的空操作指令（你的 CPU 可能希望你用**ssop** 而不是简单的 **nop** ——这要查阅手册）。

老些的 CPU 手册应当描述（具体 CPU 的）任意遇险的最大持续时间，让你能够计算必须过多少条指令的时间之后才可以保证安全。

15.3 多处理器系统和高速缓存一致性

在 1990 年代，MIPS CPU 被 SGI 和其它公司用来建造高速缓存一致性的多处理器系统。确切说，SGI 并不是这个领域的开拓者，但 MIPS R4000 是第一个从一开始设计就针对这类系统设计的微处理器，SGI 的大型 MIPS 多处理器服务器/超级计算机都是极为成功的产品。

但是，这些技术中很少是专门针对 MIPS 的，所以我们只是简单提一下。

我们这里描述的多处理器系统是一种所有的 CPU 都共享同一存储器，至少对于主要的读写存储器共享同一物理地址空间——即是一个“SMP”系统（对称多处理器系统——称为是“对称 (symmetric)”是指所有的 CPU 和存储器的关系都一样，运行时地位平等）。

对于 SMP 系统首选的操作系统结构应该是各 CPU 互相协作运行同一个 Linux 内核。这样一个 Linux 内核是为在多个 CPU 上同时执行而调整过的显式并行程序。SMP 版本的调度器为任一个调用它的 CPU 找到最合适下次运行的工作，在可用的 CPU 之间中有效地共享线程。这对从 x86 桌面上出生的操作系统内核来说是一个巨大的升级，但是多个 CPU 并行运行呈现的问题和单个多任务 CPU 呈现的问题非常相似。

再回到硬件。存储器可以构造成所有 CPU 共享的一个大块，或者构造成分布式的，其中一个 CPU 到某个存储区比别的 CPU 要“近”。但是所有的多处理器系统都有一个问题，就是共享存储器的性能比专用的低：因为总体的带宽在各个 CPU 之间瓜分，但是——通常更重要的是——管理共享存储器的逻辑给连线加入了延迟从而加大了存储器时延。如果没有大容量的高速缓存，性能将会低得极为可怕。

高速缓存的问题是，CPU 虽然很多时间都是独立运行，但它们要依靠共

享存储器互相协作和通信。理想情况下，CPU 各自的高速缓存是透明的：每个读写操作应当与 CPU 直接在共享存储器上操作产生完全相同的结果（但是更快）——这种高速缓存称为“一致的。”但是对于简单的高速缓存（正是许多 MIPS CPU 的特色），不会发生这种情况——高速缓存中拥有数据拷贝的 CPU 看不到其它 CPU 对存储器做的修改，CPU 可能修改它们自己高速缓存中的数据拷贝，而没有写回到存储器让其它 CPU 有看到的机会。

你需要的高速缓存是这样的，和通常方式一样备份内存数据，但当别的 CPU 想要访问同一数据时能自动识别并处理这种情况（多数时候只要默默地作废本地的备份就行了）。因为对于共享代码的引用相对较少，这样做效率可以很高。

要搞清楚具体怎么做得花费一点时间。目前推出的系统是这样的：存储器共享以对应的高速缓存行（通常为 32 字节）为单位进行管理。每个高速缓存行只要是读数据，就可以复制到任意数量的高速缓存里；但是一个 CPU 正在写的高速缓存行归某一个高速缓存专用。

反过来，在每个 CPU 里的实现是为每个驻留的高速缓存行维护一个明确定义的状态。总线协议的组织方式为每个高速缓存行指定一个状态机，状态转换由本地 CPU 的读写和高速缓存之间发送的消息触发。一致的系统可以通过枚举允许的状态进行分类，产生了许多术语，如 MESI、MOSI、MOESI（分别来源于状态的名称 modified、owned、exclusive、shared 和 invalid）。一般来说简单的协议导致共享成本的上升。

第一个实际的系统采用广播总线连接所有高速缓存和主存，其优点是全部的参与者都可以（同时、且以同样的次序——发现这样做极大地简化了问题）看到所有的事务。高速缓存间大多数的通信可以通过让高速缓存“监听” 高速缓存和内存间的重填和回写操作来达到。这就是为什么一致高速缓存有时叫做“监听高速缓存。”

单总线系统不易扩展到多个 CPU 或者很高频率。现代的系统使用更加复杂的高速缓存之间、高速缓存和内存之间的连接网络；这意味着你不能依赖监听，而且系统中没有一个地方让你可以找出不同请求发生的次序。情况要复杂得多...

在本书写作的 2006 年以来，这种技术正在迁移到最小的系统，这些系统中在位于一个片上系统 SoC 的多个处理器之间共享存储器。芯片级的多处理（CMP ——Chip-level multiprocessing）比你想象的更有吸引力，因为在不用很高的时钟频率下增加了计算能力。已知的唯一实用的 SoC 设计和测试方法不能交付很高的频率——而且，在任何情况下，运行于 3 GHz、功耗达 70 W 并要解决散热问题的处理器在消费类设备中基本上没有实用价值。很难在 SoC 上制造单个监听总线之类的东西。在当代（2006 年）的 SoC 中，目前的工艺技术水平是采用小规模的 CPU 集群，由一块与这些 CPU 紧密耦合在一起的逻辑来维护高速缓存的一致性，实现单个的控制点。未来的 SoC 可能会用一些松耦合的

CPU 网络，这就需要更为复杂的高速缓存一致性协议。

要让高速缓存一致的 SMP 硬件能工作，还能高效地工作非常困难。在这些努力过程当中，产生了相当多的如何才能做好的口口相传的经验体会。在与象 Linux 内核这类健壮的、多 CPU 应用打交道之前，这些东西值得介绍一下。下面是一些常见的头条问题：

- 选择不需要一致性管理的页：高速缓存一致性由自己处理，但是如果操作系统可以告诉硬件那些页不用共享或不要一致性管理，那么性能会更好。只读页不需要管理——它们来自内存，高速缓存可以根据需要复制任意多份。当然没有页面是完全只读的：总会有某个时候在某个地方写入数据。但是 Linux 中，指令页足够接近只读，值得特殊处理。内核自己的指令在任何高速缓存之前已经安全写入，所以没有问题。应用程序的指令页通常是从文件读入的，在读入文件这一点，要作废所有高速缓存的拷贝；但是过了这点以后也没有问题。

  许多数据属于单个线程。进程的栈总是如此，单线程应用程序的所有用户空间数据也是如此。不幸的是，虽然在任意一个时间点只有一个 CPU 对单线程的进程感兴趣，但是应用程序在重新调度时可能从一个 CPU 迁移到另一个 CPU 上（如果想要多处理器系统工作得好，就要对迁移做一些约束）。对于已知的单线程数据页放松一致性管理，但在线程迁移时是需要做一些容易出错的手工清理高速缓存的工作，这样做的好处是否值得，还是一个需要判断的问题。

- 原子性、临界区和多处理器锁：如果不注意保证数据读写在第 14.2 节意义下的原子性，那么多个 CPU 运行同一代码并对共享数据结构访问进行协商的 SMP 系统是无法工作的。

  MIPS 体系结构的 **ll/sc**（连锁加载和条件存储）指令是构建原子操作和锁的基本原语，其设计得可以很好地扩展到大规模多处理器。

- 存储器访问顺序不确定和存储器防护：在高速缓存一致的多处理器上，当任一个 CPU 写入存储器地址时，另一个正在读取同一地址的 CPU 会看到（也许晚一点）数据更新了。但是一旦你离开围绕单个通用总线构建的系统，很难保证读写能保持相对次序。大多数时候次序可能没有关系——运行于不同 CPU 上的线程反正不得不面对不知道其它线程进展的事实。但是用共享存储器在线程间通信的软件总是非常脆弱的。

  在第 10.4 节讨论过这个问题，该节描述了 MIPS 的 **sync** 指令作为存储器防护的功能。

  sync 指令还有另外一面。在许多 CPU 里，它还有针对具体 CPU 的额外的或更强的语义，比如“全部数据都已离开系统接口，”或者“一直等到我的整个读写队列为空。” 需要阅读你的 CPU 手册找出来。

15.4 关键例程的手工调优

值得很快地看一些优化的 Linux 代码，来感觉一下针对具体体系结构或者 CPU 的性能优化的价值到底有多大。

clear_page() 例程在 Linux 内核使用得很多。全部填满零的页面就是直接满足应用程序数据空间“未初始化”部分的需要的，而且也用作一种清除上个程序页面以防从一个程序到另一个程序之间偶然的信息泄漏（这种泄漏违反安全策略）。

这里实现的 clear_page() 采纳了几个思想。它展开循环——每次循环清除一整个高速缓存行（在这里是 32 字节，8 个字）。

该例程也用了 MIPS 特有的 **pref** 指令以免等待高速缓存。如果预取真的需要从存储器读取数据，比起 CPU 执行速度那将花费很长时间：所有我们要提前很久就做预取。在这里，“很久”就是 512 字节，16 个高速缓存行。

但是如果 CPU 可以理解预取提示，就用专门的“为写预取”版本，可以无需读进任何数据而新建一个高速缓存行（程序员要保证重写整个高速缓存行）——参见第 8.5.8 节。

下面是该函数，增加了些注释。记住 PAGE_SIZE 通常是 4096。

```
#define PREF_AHEAD 512

clear_page:
    # the first loop (main_loop) stops short so as not to prefetch off
    # end of page
    addiu   a2, a0, PAGE_SIZE - PREF_AHEAD

main_loop:
    # the prefetch. Bring in cache line, but with luck we won't read
    # memory. But if all this CPU offers is a simple prefetch, that
    # should work too.
    pref    Pref_PrepareForStore, PREF_AHEAD(a0)

    # now we're going to do eight stores
    sw      zero, 0(a0)
    sw      zero, 4(a0)
    sw      zero, 8(a0)
    sw      zero, 12(a0)
    addiu   a0, a0, 32    # some CPUs choke on too many writes at
                   # full-speed, so increment the loop pointer
                   # in the middle to give it a break.
    sw      zero, -16(a0)
    sw      zero, -12(a0)
```

```
    sw      zero, -8(a0)
    bne     a2, a0, main_loop
    sw      zero, -4(a0)  # last store in the branch delay slot

    # the second (end) loop does the rest, and has no prefetch which
    # would overrun

    addiu   a2, a0, PREF_AHEAD
end_loop:
    sw      zero, 0(a0)
    sw      zero, 4(a0)
    sw      zero, 8(a0)
    sw      zero, 12(a0)
    addiu   a0, a0, 32
    sw      zero, -16(a0)
    sw      zero, -12(a0)
    sw      zero, -8(a0)
    bne     a2, a0, end_loop
    sw      zero, -4(a0)

    jr      ra
    nop
```

第 16 章　Linux 应用程序代码、PIC、库

运行应用软件的 GNU/Linux 线程可以合理地称为*程序*。大多数这类应用在加载时将之前相互独立的部分拼接在一起，这个过程有赖于一种相当激进类型的位置无关代码（position-independent code，即我们标题中的 PIC）。在我们开始这个题目之前，要注意的是 Linux 内核并没有强制你用任何特定的方式编译和构建应用程序的二进制代码：内核对于用户空间的事情几乎没有做任何假定。

典型的 GNU/Linux 程序包括大量的独立编译的库代码，这些库代码在多个（运行相同或者不同的程序的）进程间共享。让应用程序的“主程序”在程序运行时*动态*链接到库代码，而不是在编译时将库代码构建进去形成一个独立的整块的二进制文件，这种做法具有很多优势：这里只说两点，动态链接极大地减少了程序二进制代码在磁盘和内存中占用的空间，还意味着修正库中的缺陷只要提供一个新的库文件就可以了。

磁盘上的程序和库位于*目标文件*中，其中包含预先编译好的二进制代码、初始化的数据和一些将程序和库拼接到一起时必要的管理信息（包括符号表和重定位信息）。

图 16.1 是一个典型应用程序的虚拟存储映像的示意图。

程序由多个准独立的*链接单元*[1]构成——主程序一个，每个用到的库各一个。

每个链接单元加载进若干内存块（段(segment)）中，其中许多都有由目标文件块（区(section)）提供的初始数据。但是有些内存块——特别是容纳堆栈的那些内存块——是由加载程序创建的。

在 Linux 里，在包含同一链接单元的各个活动的程序之间（按页）共享程序代码。参与共享的程序各自可以拥有完全不同的存储器映射；不管代码位于虚拟存储器的什么地方，代码自身的内存映像都必须能够正常工作。换句话说，代

[1] 对于构成动态链接程序的单个二进制的文件还没有一致的叫法。我们称之为“链接单元(link unit)”的东西曾经被称为动态共享对象（DSO(dynamic shared object)）、目标、或模块。但这与 C++ 的对象（object）没有关系；“模块”这个词已经用来表示编译器在一次构建中生成的东西。所以本节我们使用这个中立的词“链接单元”以提醒你这个二进制代码是在链接的最后阶段生成的东西。库链接单元文件通常用“`.so`”后缀表示。

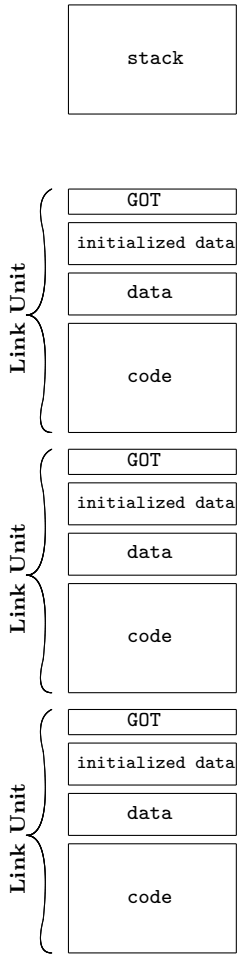

图 16.1: Linux 程序内存映像

码必须是位置无关代码即 PIC。仅凭分支和调用指令采用相对 PC 寻址这一点就把代码称为“位置无关”的现象并不罕见。但是 Linux PIC 代码的位置无关性要比这彻底得多，因为不光代码连数据也可以位于任何地方。不管链接单元的段最后位于存储器的什么地方，所有的代码和数据引用都必须继续有效。

随之而来的就是，当一个链接单元的代码从另一个链接单元中调用一个函数或者引用一块数据时，函数或数据地址（当前的实际值）必须由正在运行的程序计算出来。这就要有 GOT(global offset table 全局偏移表)。

每段代码都有一个 GOT，在目标文件中定义。在动态加载器完成加载时，GOT 包含一个指针[2]，指向该链接单元的代码用到的每个外部函数或数据结构。链接单元的目标代码对这些外部地址一无所知：它只有一个符号表，通过符

[2]这种说法过于简化了。只要允许链接单元的代码调用外部函数或引用外部数据，就可以把任何东西都放进 GOT。

号和索引间的映射跟踪 GOT 的入口。你常常会看到这种说法，说 GOT 为每个外部符号保留一个入口项——这个说法没错，但在你查看加载进内存的二进制代码时容易引起困惑。动态加载器读取两个链接单元的目标文件，为它们分配空间，然后通过匹配符号来找到正确的外部地址放进 GOT。

我们刚才说过，内核并不关心应用程序是怎样构建的，事实上动态加载器并不是内核的一部分。它自身也是一个链接单元，主程序就象引用其它动态库一样引用动态加载器。共享 PIC 库的应用程序启动时先运行动态链接器，将虚拟机 VM 映像的其余部分拼接到一起，然后才调用应用程序的 `main()`。

16.1 链接单元怎样构成一个程序

把一个新的链接单元粘合到程序里头有三种方式：

- 当程序加载时带入：每个链接单元，从主程序（可执行文件）开始，包含一个在调用 `main()` 之前要加载进虚拟存储器的其它链接单元的列表。这并不是说程序实际上读入了存储器，甚至也不是说已经设置了虚拟存储器映射：这还是一个虚拟存储的操作系统，程序代码和数据的大部分直到使用时才会出现。

- 首次引用例程时加载：动态加载器推迟对某些链接单元的加载（“懒联编(late binding)”），这种做法的前提是链接单元是一个纯粹的库，其接口完全由函数调用构成，没有外部可见的数据。

  加载器是这样做的：设置与懒联编的链接单元相关的 GOT 入口，由链接单元回调动态加载器中的一个函数——该函数负责加载缺失的链接单元、修正 GOT 并且（小心地）将控制返回到新加载的链接单元的函数。在目标文件中要标识出懒联编的链接单元。

- 直接加载：用 `dl_open()` 函数找出库链接单元，然后粘连进程序，并构建一个内容包括指向新程序要输出的函数的指针在内的数据结构。

  这样一个链接单元的工作方式很象一个 C++ 对象（函数指针让人回想起 C++ 的方法）——这就是问什么链接单元有时被叫做动态共享对象的原因。

大多数时候，加载不同链接单元的次序并不重要：加载次序会影响存储器布局，但是每个链接单元不管在什么地方都能工作。

但是有可能构建出这样的程序，其中不同的链接单元提供了同样的符号。[3] 当发生这种情况时，首先加载的链接单元“胜出”，提供变量。

[3]有时这只是个错误，会导致链接失败，但是有些符号特意标识为可有多个带有安全限制的定义（加载时第一个定义胜出）。

16.2 全局偏移量表（GOT）的组织

程序构建过程（我指的是编译和生成程序时的链接）为每个链接单元至少定义一个 GOT，当然超过一个是完全合法的。[4] 每个函数可以找到自己的 GOT，因为 GOT 的位置离函数入口点的偏移量是已知的（记住，链接单元作为一个整体载入内存，所以内部偏移量与编译时一样）。

GOT 的每个入口都是——等效于——指向一段数据或者代码入口点[5] 的绝对指针，由使用该 GOT 的链接单元中的函数（直接或者隐含地）定义为外部指针。GOT 项依赖于链接单元在程序地址空间内的分布，由动态加载器在链接单元被加载时计算。

因为链接单元的代码（和只读数据）的二进制映像在 Linux 之类的操作系统中是真正共享的，我们不能在加载时对代码进行任何依赖于地址映射的修正。但 GOT 则属于数据的一部分，而且每个程序自己都有一份拷贝。GOT 中的指针对使用共享链接单元的每个程序可以各不相同。

GOT 访问很频繁，所以在 PIC 代码中，编译器/链接器在合成代码时保持 **gp** 寄存器指向每个函数的 GOT。采用寄存器 **gp** 作为表的基址是个不错的选择，因为这种用法类似于采用相对 GP 寻址数据的想法，后者使得 **gp** 成为了保留寄存器。老式的相对于 GP 寻址的数据和 PIC 本质上就是不兼容的，所以不会有大的冲突。

GOT 指针必须在每个函数开始时计算出来并加载到 **gp** 寄存器。这点不太好好办——对于 PIC 代码，编译时我们不知道 GOT 会在哪里。但是 GOT 是同一链接单元的一部分，所以从函数自身的入口到 GOT 的偏移量在编译时是已知的。

如果一个远程函数“知道”自己被用地址调用过，那么这个计算就会容易些，因为就会有一个寄存器已经包含了该函数的地址，计算指向自己的 GOT 的指针时就省点事。Linux/MIPS 约定要求所有的 PIC 调用采用 **t9** 来容纳被调用函数的地址。

然后函数首部就可以单步（不一定是单个指令）算出 GP：

```
entrypoint:
    addu    gp, t9, GOT - entrypoint
```

用 GOT 入口访问不同链接单元数据的代码仅仅知道应该到哪个 GOT 槽中去查找指针：在链接单元链接到一起的时候分配 GOT 槽，但是存储在 GOT 中的指针直到程序加载时在其它链接单元装入后才确定。所以编译器必须生成执行双重间接寻址的代码，加载指向外部数据的指针。

[4] 使用多个 GOT ——通常是每个 C 模块一个——是处理超过 128 KB 的 GOT 的大链接单元的首选方法（如果 GOT 超过这个限制，编译器和汇编器不得不生成更长的代码序列从 GOT 检索指针）。

[5] 在某种限制的情况下，编译器可能“知道”两个数据项之间的距离：那种情况对两个数据用同一个 GOT 入口。

```
load:
    lw  t1, gp(MYSYMBOL_INDEX)
    lw  t1, (t1)

store:
    lw  t1, gp(MYSYMBOL_INDEX)
    sw  t2, (t1)

call:
    lw  t9, gp(MYFUNCTION_INDEX)
    jalr t9
```

PIC 代码比用固定地址的代码跑得慢，但是共享库的优势更多，足以抵消为此付出的代价。我们在本章开头的时候只提到了 PIC 的两点优势，那只是为了提起你进一步了解的兴趣。这些优势也就是一般懒联编带来的典型优势。还可以列出更多的优势，包括：

- 缩小的二进制代码：如上所述，你的系统的二进制代码不需要复制 C 库函数。庞大复杂的库常使简单的应用程序相形见绌。
- 修正一个库中的缺陷就修正了所有依赖的应用程序：系统中最复杂的一些代码位于库中，所以即使单个的系统可能安装了不同的应用程序集合，让库以与单个二进制文件相同的周期更新仍然有好处。
- 为提供遗留接口添加了新的选项：当你想要淘汰某个库函数或者内核函数时，你可以提供一个采用新的特性重新实现的兼容的库。
- 库在应用程序和内核之间提供了一个隔离层：这让应用程序和内核可以各自单独演化。

最后，动态库代码比静态链接的 C/C++ 提供了一种更高级的开发环境。

除了通过 GOT 做各种外部调用和数据引用所带来的执行时间开销之外，动态链接库还有一个不利的因素。有了动态加载，应用程序和库就陷入非常复杂的依赖关系网之间而纠缠不清：这使得安装新软件时容易出问题，删除系统中的旧软件就更容易出问题。一个健壮的动态库检索和版本标识系统是必不可少的。早期的 Windows 系统曾经受到臭名昭著的“DLL 地狱”的困扰，就是因为不同的应用程序对同一个库的要求互不兼容。

现代的系统（Windows 和 Linux 之类）似乎构造了一个能够稳健跟踪依赖关系的依赖数据库系统。动态链接的代码还会一直存在下去，从存储空间的经济角度来说，它对于带有大量应用软件的任何嵌入式系统都很重要。

附录 A　MIPS 多线程

多线程硬件是 2004–2005 年间热炒的计算机界的“下一个重大突破(NBT——Next Big Thing)。”到了 2006 年不再那样：虚拟化看上去好像成为了该年度的 NBT，该技术上次曾在 1975–1980 年间获此殊荣。多线程炒作的热潮虽然已经过去了，但多线程技术本身的重要性仍然不曾减退。

所以我们试图定义一下什么是多线程，然后问它有为什么有用。我们用“多线程(multithreading)”时不加连字符，用缩写“MT”代表 MIPS32/64 的体系结构扩展（MIPS ASE）。

A.1　什么是多线程？

线程就是按照程序员意图顺序执行的指令序列。

这个定义看上去真的有点简单，但却让“线程”成了一个可能相当棘手的东西（沿着子程序调用树后是各种纷杂的路径，循环中还会有嵌套的循环）。但是当一个传统的 CPU 处理中断的时候，那就不是同一个线程了（至少，在中断处理程序返回之前不是）。操作系统称之为“进程”、“任务”、“作业”等等的东西都是不同的线程。

当你想要多于一个线程的时候才需要定义线程（否则线程就是 CPU 正在运行的程序）。容易看出为什么在有许多不同的用户都急着想要运行他们的程序的分时计算机上，你想要多个线程。

有时你想要计算机运行单个的程序，但是必须协调响应性的需求。有些问题拥有内在的本质上的并发性。（理论上）你可以写一个程序，应用某种多路选择调用来等待若干事件、找出事件并且调用相应的线程例程。或者你也可以发明一个并发程序设计语言。

但是直接并发的程序很难写，从来没有一种并发语言真正得到广泛应用。在 1970 年代，关于设计、编写和测试处理并发事件的程序实用性的研究达成了一个结论。研究者最终停留在采用通过简单固定的接口而互相影响的多个简单线程的思想上。

对于这些接口是应当采用基于消息传递还是基于更简单的类似信号量的机制爆发了一场异常激烈的两败俱伤的内战：现在各方早已放弃了战争，结局是大

家共赢。现代的操作系统包含了许多不同的通信机制，当然不同的程序员可能更习惯于采用不同的子集。

线程这个词作为区别于“进程”这个术语出现，是为了让线程不再有其它负担：UNIX 进程每个都拥有自己的地址空间，但是线程则可以共享地址空间。POSIX 标准 1003.1（称为“PThreads”）为显式共享代码和数据的线程推荐了一组丰富的跨操作系统的编程接口，PThreads 已经得到现代操作系统的广泛接受。

A.1.1 同时运行两个线程需要哪些资源？

当然两个完整的 CPU 可以做到。但是这个要求好像太多了：我们想问的是最少需要什么？

通过查看（运行于一个传统 CPU 上的）多线程操作系统为每个线程保持的信息，可以得到对这个问题答案很好的提示。当线程不在运行时，察看线程，可以找到一个 PC（线程下次开始时要执行的地址）和保存的所有程序员可见的通用寄存器的值。对于 MIPS 上的用户特权级程序，就是通用寄存器和乘法器的 **hi/lo** 累加器。操作系统需要持有当前正在运行的那个线程的标识符，以及当前是否持有内核特权级（当执行系统调用时就会这样）等标志。维护多个地址空间的 MIPS 操作系统也会需要保存 ASID 值，因为当转换地址时硬件要用到。

基本上就这些了：PC、通用寄存器、线程标识符、核心态、ASID。操作系统称之为线程环境，在最小情况下，每个线程必须自己有这些寄存器和域的拷贝。

你还可以从另外一个角度来看。假如你想要某人购买你的多线程硬件，要是你的硬件能够运行现有的软件那将很有说服力。双处理器 x86 PC 的硬件（尤其是与其紧密相关的 x86 小型服务器硬件）已经出来好几年了，Microsoft Windows、Linux、以及其它 Unix 已经知道怎样利用那种硬件了。所以当 Intel 在其“超线程”CPU 中推出多线程能力的时候，它增强了硬件的能力使得只有一个多线程 CPU 的系统能够不用修改就可以运行为双 CPU 系统构建的软件。要在 MIPS 上实现这一点，你需要为每个“虚拟处理器”单独提供各个寄存器的拷贝。

那是两种不同的多线程方法。MIPS 应当采用哪一种？在我们回答这个问题之前，让我们先补习一下为什么值得这样做。

A.2 为什么要用 MT？

2005–2006 年间的传统 CPU 撞到了“存储器（时延的）绝壁(memory wall)。”运行一条指令花费的时间降低的速度已经比访存时间降低的速度快了很多。快多少呢？Intel 的 32 位 x86 CPU 已经从 1985 年的 16 MHz 到了 2005 年的大约 3 GHz ——20 年快了 200 倍。在同一时间，存储器访问时间也下降了，但是仅仅

四倍，从 1985 年的 180 纳秒到现在的 50 纳秒。[1]

粗略的说，这意味着在 1985 年读取一次内存让 CPU 要停下等四个周期，而现在要让 CPU 停下等 150 个周期。高速缓存和让返回数据之前尽量提前执行的 CPU 设计技术等方面的大幅度改进掩盖了这个问题，但是进一步提升的空间极为有限。因为要等待存储器的数据而让 CPU 超过一半的时间在空转，这种情况已经司空见惯。

嵌入式处理器不会运行于 3 GHz 那么离谱，但是它们的高速缓存和 CPU 设计都简单。当代的嵌入式系统，也差不多让 CPU 有 50% 的时间在等待。

多线程硬件对嵌入式系统有用还有另外一个原因。如果你有一些要求很高的 I/O 服务需求，用一个专门的线程来服务这个需求能实现更快的响应。你可以把从 I/O 系统读取数据的线程“停放”在那里，数据一准备好该线程就跳起来工作。没有中断的开销，而且——更重要的是——没有潜在的操作系统开销。谁说 I/O 线程必须运行标准的操作系统代码？

所以说多线程是个好事情。我们该怎样做呢？价钱成本是否也不错呢？与通常嵌入式系统一样，影响成本最大的因素是芯片面积和功耗。

A.3 MIPS 怎样实现多线程？

RISC CPU 喜欢把策略决定留给软件负责，而不是交给硬件。RISC 的原则倾向于灵活通用的解决方案。这使得 MIPS MT 有以下几个特点：

- 互相可见线程寄存器：这给了操作系统对多线程的完全控制。**mftr** 和 **mttr** 是特权指令（“与线程寄存器传送数据”）允许操作系统软件掌握了解不同 TC 的寄存器。这是所有跨线程初始化和维护的基础。

- 启动时为单线程：这点遵循了引导软件最少差异的原则。操作系统或者引导软件在准备好后可以唤醒更多线程。

- 同时提供两种形式的多线程：最小开销线程引擎和完全的“虚拟 CPU”可以混合匹配。基本的 MT 特性就是最小线程引擎，称作 TC（来源于 Thread Context）。如果我们就停留在这一步，CPU 中除了直接的线程环境外其它所有的东西都是共享的。

  其实，我们允许 MT CPU 复制一个 MIPS32/64 兼容的 CPU 所需要的一切。这些类似 CPU 的寄存器组和其它资源的每一个叫做 VPE：一个或多个 TC 和 VPE 相关联，并共享它们的寄存器和资源，VPE 中的 TC 构成一个“虚拟处理单元(virtual processing element)，”这就是 VPE 缩写的来历。一个 CPU 可以实现多个 VPE 以生成看上去像多个 CPU 的东西。

[1] 存储器带宽上升的速率要快得多。但正如所说的“花钱可以买得到带宽。要想缩短时延，除非收买上帝。”——技术就相当于这句话中的钱。

MIPS 公司第一个 MT 产品，34-K CPU，最多可以有五个 TC 在两个 VPE 之间任意分配。

VPE 间可以共享很多东西，当然每个 VPE 从软件上等价于一个 MIPS32 CPU：这不是真的双 CPU，只是看上去很象。高速缓存、主流水线、控制逻辑、算术逻辑单元和系统接口都是共享的。TLB 在不同 VPE 间可以用硬件隔开，或者共享（共享的 TLB 版本涉及对操作系统的许多局部修改）。

机器运行的每个指令都有一个 TC 号，用来选择具体的环境。当该指令访问某些状态的时候——例如读写通用寄存器的时候——就用其 TC 号来扩展已经定义在指令内的寄存器号域。一条指令根据不同的 TC 号看到的是一组不同的寄存器集：非常简单但是的确有效。

由于上面提到的 TC/VPE 技巧，MIPS MT 不是那么简单。这条指令可能是 TC #5 的（从第五组通用寄存器集访问数据）或者 VPE #1（从第一组获得 CP0 寄存器）。这个应当没问题。当然更为复杂的是，让那些不能简单的归为寄存器的 CPU 资源工作。但是那已经不属于体系结构，而是属于具体实现了；这些材料你必须阅读 CPU 手册。

MT 规范没有硬性规定，但是实际的 MT CPU 需要能快速切换线程，否则线程切换损失的时间可能会吃掉潜在的吞吐量增加。其它东西都一样，通常最好在尽可能细的粒度上混合线程：在单发射 CPU 中，这等于说只要多个线程准备好，每个时钟周期从不同线程各发射一条指令。

A.3.1 MT 新增的 CP0 寄存器

这些寄存器可以分为三组：提供给每个 TC 的、提供给每个 VPE 的、还有几个是提供给每个 CPU的。最后一个是提供 CPU 的资源清单（多少个 TC? 多少个 VPE? 等等）的寄存器，可以共享；这里不进一步说明。

每个 TC 的寄存器包括：

- TCHalt: 一位的寄存器，写入 1 让目标停机。目标 TC 停机时，其状态是稳定的，可以安全写入它的寄存器。
- TCRestart: 线程“PC” ——当线程停止时，这就是下次运行时第一条指令的地址。当线程不是停止状态时，没有太多的意义。
- TCStatus: 各 TC 的“遗留”域——内核/用户状态、ASID、以及指令集选项标志（比如使能浮点的标志等）。也有一个标志表明线程停止的指令位于分支延迟槽（在这种情况下，**TCRestart** 指向前面的分支指令）。
- TCBind: 包含与该 TC 关联的 VPE 的 ID（在 34-K 族 CPU 中可写），以及该 TC 的 ID，后者是只读的。

- TCContext: 一个纯粹由软件读写的寄存器，典型的用法是存放特定操作系统的线程 ID。

各 VPE 的寄存器 **VPEControl** 用于操作系统运行过程中可以合理改变的控制域，而 **VPEConf0–1** 的配置域极可能是一次设置后就再也不变了。

A.3.2 异常和中断

单线程的 MIPS 体系结构的 CPU 的异常通常对流水线是极具破坏性的，常用的实现方法丢弃了许多执行状态（流水线清空，预取的指令丢弃）。MIPS MT 机器上的异常发生在线程环境中——其它的线程（至少那些在别的 VPE 上的线程）可望不受打扰继续运行。所以可以预计当我们在多线程机器上重新定义线程时会有困难。

心中要记住操作系统是一个程序（其实是一组线程）。具体哪个 TC 执行哪一部分可能关系不大。如我们在本附录的开头看到的，每个异常处理程序自身就构成一个线程。所以要运行一个异常处理程序，我们需要借用一个 TC 来中断其原来的线程并运行这个异常处理线程。

异常分两种类型。中断是异步的——它们的发生原因与具体指令无关。但是大多数其它异常是同步的——它们与特定的指令相联系。我们先看看后一种类型。

同步异常处理程序由导致异常的指令的 TC 运行。TC 立即停止正常线程上的工作，开始从适当的异常处理程序取出指令执行。

MIPS MT ASE 规定一旦 TC 进入异常状态，同一 VPE 内所有其它 TC 一律暂停。直到异常处理程序离开异常模式，其它 TC 的指令才可以执行：就是说，要一直等到 **SR(EXL)** 位被异常处理程序结尾的 **eret** 指令清零，或者因异常处理程序分支到操作系统较少受限制的代码部分而清零。异常处理程序尽量减少花费在异常态的时间本来就是良好的做法，大多数操作系统都是这样做的。

但是如果你的应用程序需要最大化并发性，你应当考虑最小化异常——你可以用一个在 ITC 访问或者 **yield** 条件上阻塞的线程。当然安排异常处理程序（尽可能早地）保存必要状态以便退出异常模式。

A.3.3 MIPS MT 和中断

在 MIPS 体系结构中，中断管理通过 CP0 寄存器（具体就是 **Cause** 和 **SR**）进行。这些寄存器为每个 VPE 而不是每个 TC 复制一份，所以中断屏蔽和传递由每个 VPE 管理。甚至硬布线连到核的中断信号也是由每个 VPE 处理。

每个中断输入可能只连到一个 VPE 上，也可能连到全部 VPE 上，所以哪个 VPE 接收中断涉及到系统的硬件布线方式，但也可能和软件配置有关。如果你不加屏蔽地把一个中断连接到多个 VPE 上，任意几个 VPE 都可能响应中断异常——你大概不希望发生这种情况，所以要么别连线要么别使能某些中断。

中断异常可以由与 VPE 相联的任何可用的 TC 响应。

MIPS 体系结构已经提供了多种方式来拒绝中断异常。异常模式、全部中断使能标志（可能为零）和单个中断的屏蔽位，即 **SR(EXL)**、**SR(IE)**、**SR(IM)** 位——这里列出的不是全部——都可以防止该 VPE 上的线程发生中断。

MIPS MT 规范增加了又一个不响应中断的理由。现在你可以设置一个新的每个 TC 的 CP0 寄存器域 **TCStatus(IXMT)**（中断豁免），该域可以防止特定 TC 被用于中断异常。

A.3.4 线程优先级提示

有些应用程序开发人员对于可以控制 CPU 带宽偏向一个线程而不是另一个线程的能力表示感兴趣。MIPS MT 规范提供了这个特性，尽管并没有说明怎样实现。一个调度策略管理器（policy manager，常缩略为 PM）是 CPU 外部的一块逻辑，可以针对具体应用定制。PM 为每个 TC 生成一个 2 位的优先级。在核里面，总是优先执行可运行的高优先级 TC，而不是可运行的低优先级 TC。

现在就来评论开发人员采用这个方案的成功性还为时过早。

A.3.5 用户权限的动态线程创建——fork 指令

多线程硬件还有一个很有意思但还没有商用化探索的应用，就是提供另外一种发现及利用串行算法的并行性的方法。例如，一个循环可以通过两个线程分别运行其奇数和偶数次循环来优化。

如果这要在带保护的操作系统的支持下完成，就要求有一个非常高效的机制即需发射新线程。MIPS MT 的 **fork** 指令提供了这样一种机制。

这样 **fork d,s,t** 成了一条用户级指令。如果一切正常，该指令在准备就绪的 TC 上开始一个新线程，从 **s** 所指向的指令处开始执行。子线程的 **d** 获得父亲线程的 **t** 值。

操作系统通过维护一个分支就绪但是此外空闲的 TC 缓冲池（通过一个标志位 **TCStatus(DA)** 来识别）。当新线程完成应用程序的任务后，就终止执行，并用一条 **yield $0** 指令把 TC 交还给缓冲池。[2]

fork 是一个前瞻性的特性：在本书写作时，还没有操作系统支持这样的 TC 缓冲池。放在那里是为了让 MIPS MT 体系结构能够随着多线程概念的传播占据有利位置。

A.4 MT 在实际中的应用

现在依然还早，但是人们用多线程，具体到用 MIPS MT 在做些什么呢？

[2]给 **yield** 的源寄存器操作数一个非零的值则用于完全不同的目的。

泛泛地说，人们正在探索或者开发的有两种不同的系统。一种是寻求修改标准的 SMP 操作系统代码（也许很少的修改），以适合于向相对传统的多线程工作提供 MT 硬件多线程的高效率；另一种寻求在操作系统“之下”为全新的应用提供一个底层、超快速响应的“实时”环境。

A.4.1 SMP Linux

移植一个 SMP Linux 到包含两个或多个相当于 SMP 系统的 CPU 的 TC 的 MT CPU 上是有可能的。如果你有一个已经用到了 Linux 实现的 PThread 线程 API（称为 NPTL）的应用程序，那么你把该应用程序放到这样的 Linux 上运行就能利用多线程功能了。

SMP 系统需要在 CPU 间工作的锁和信号量，对于 MIPS 系统，这些是基于连锁加载/条件存储指令对（第 8.5.2 节介绍的**ll/sc**）构建的。所以除非 MT CPU 在 TC 间保持 **ll/sc** 的语义不变，否则不能用这个方法：MT CPU 确实保证了在 TC 间这两条指令原来的语义不变! 34-K 族的硬件跟踪每个 TC 发送的 **ll** 所用的地址，如果其它 TC 的存储操作修改了共享同一个双字地址就认为是破坏了连锁，这样保持了 **ll/sc** 语义不变。如果某些管理软件接管 TC 并写入 **TCRestart** 寄存器执行重新调度的话，连锁位也会打破。

如果 TC 位于不同的 VPE，那么 SMP 移植就很容易，因为一个 VPE 中的单个 TC 设计的本身就像是一个独立的 CPU。但是 MIPS 公司也证明了可以在同一个 VPE 中用不同的 TC。这里有一点难度，但是看上去不会导致太多的性能损失。

然而很重要的一点要注意，只有一部分应用程序才会直接受益。你需要一个使用 Linux 环境、利用了线程的应用程序，而且在大多数时间不仅有两个线程在运行，还要求在两个线程间基本上平均地分配了 CPU 资源。

A.4.2 用 MT 实现高速响应的程序设计

MT 的先行者采用的另一种方法是用专用的 TC 提供接近硬件的能力，其响应时间可以独立于操作系统内发生的运行于别的 VPE 上另一个 TC 中的任何事件。这样一个系统有希望在两个方面都达到最佳结果：对于高层代码提供更方便的编程环境，对于事件的快速响应让底层硬件变得更简单。

这种系统上的低层代码极为需要高效的同步机制。比起处理器的执行速度，即使是 I/O 读取的速度也慢得像蜗牛爬行。MIPS MT 定义了两种机制：信号让与(yield-on-signal)和门控存储*(gated storage)*。

上面提到说 **yield** 指令具有双重应用：当它不是用来提供与 **fork** 相反的用户级线程的终止时，该指令提供了一种让线程等待硬件事件的方式。硬件事件根据 CPU 取样到的外部信号有效性来定义。

34-K CPU 可以取样任意 15 个外部信号。**yield** 指令一直等到所选择的其

中一个信号有效，选择哪一个信号取决于一个输入参数寄存器的值，该寄存器的值被解释为一个 15 位的位向量：位为“1”表示响应相应编号的输入信号。当其中一个信号有效时，在一条 **yield** 指令上等候的底层线程立即开始执行。

但是你可能想要一个还能传输数据并且在软件线程之间可用的同步机制。如果是这种情况，门控存储器接口就派上用场了。MIPS MT 定义门控存储器为一个特殊的物理地址区域，在其中的读写指令会导致阻塞。读写门控存储器的线程在该读写操作上阻塞，一直到数据成功传输完毕。还可以强行终止未完成的操作（让每个等待的线程接收到一个异常）。

在高速数据流可能会被错误或数据块结尾条件中断的硬件接口中，门控存储器接口比较适合。

34-K CPU 设计和一个称为 *ITC*（线程间通信存储器 (interthread communication memory) 的缩写）的门控存储器一起授权：ITC 提供了采用“空/满（empty/full）”存储或者短 FIFO 的软件对软件的通信。更多的信息，请参阅 MIPS 公司的 CPU 手册。

附录 B　MIPS 指令集的其它可选扩展

B.1　MIPS16 和 MIPS16e ASE

MIPS16 是一个可选的指令集扩展，它能把二进制程序的尺寸减少 30-40%。MIPS16e 是已经用在 MIPS16/32 标准的 CPU 上的原始 MIPS16 指令集的一个略微增强的版本的名字。MIPS16 定位于代码尺寸作为主要考虑的场合——多数情况下这种场合就是指极低成本的系统。尽管只应用于特定实现，它却是一个多厂商标准：MIPS 公司、LSI、NEC 和 Philips 都生产支持 MIPS16 的 CPU。

在前面 1.2 节中我们说过，使 MIPS 二进制代码比其它体系结构大的原因并不是 MIPS 指令完成的工作少了，而是因为指令本身更大了——每个指令四字节，相比之下某些 CISC 体系结构平均指令长度只有三个字节。

MIPS16 增加了一种模式，在这种模式下 CPU 可以对固定长度为 16 位的指令进行译码。大多数 MIPS16 指令展开后成为一条正常的 MIPS32/64 指令，很显然这只能是一个相当受限制的指令子集。关键就在于使得这个子集能高效编码的那部分程序足够大，以使程序的总体大小得到显著的压缩。

当然，16 位指令并不能构成一个 16 位指令集。MIPS16 CPU 是带有 32 位或者 64 位寄存器的真实 MIPS CPU，其运算也都在整个寄存器上操作。

MIPS16 远不是一个完整的指令集——比如它既没有 CPU 控制指令，也没有浮点运算指令。[1] 但这没有关系，因为每个 MIPS16 CPU 也必须要运行完整的 MIPS ISA。你可以运行 MIPS16 和正常的 MIPS 指令混合的代码。每个函数调用或者寄存器转移指令都能改变运行模式。

在 MIPS16 中用指令地址的最低有效位对模式编码是既方便又高效的。MIPS16 指令必须偶字节对齐，所以第 0 位不再是指令指针的组成部分了；取而代之的是，每条跳转到奇数地址的指令开始执行 MIPS16，每条跳转到偶数地址的指令回到正常的 MIPS。MIPS 子程序调用指令 **jal** 的目标地址总是

[1] 对部分指令集提供将指令长度压缩一半的版本，这个想法不是 MIPS 发明的；ARM 公司的 ARM CPU 的 Thumb 版本是首先提出这个想法的。

字对齐的，所以引入了新指令 **jalx**，它隐藏了指令中的模式转换。

为了把指令压缩到一半大小，对于大多数指令我们只分配了 3 位来选择寄存器，这样只有 8 个通用寄存器可以自由访问；在许多 MIPS 指令中可以见到的 16 位常数域也被压缩，通常压缩成了 5 位。许多 MIPS16 指令只指定两个寄存器，而不是三个。另外，还有一些特殊的编码规则将在下一节描述。

B.1.1 MIPS16 ASE 中的特殊编码格式和指令

MIPS16 有两个特定的弱点会加大程序尺寸：5 位的立即数域构造常量是不够的，在 load/store 操作中也没有足够的寻址范围。三条新增指令和一种特别的约定有助于解决这些问题。

extend 是一条特殊的 MIPS16 指令，它由 5 位的代码和 11 位的域构成。这个 11 位的域可以和后续指令中的立即数域相连接，这样就允许使用一对指令来编码 16 位立即数。这条指令在汇编语言中看起来就像一个指令前缀。

装载常量在正常的 MIPS 模式下都需要额外的指令，在 MIPS16 模式下更是巨大的负担；把常量放在内存中然后再读它们会更快一些。MIPS16 增加了相对于指令自身位置的加载操作(PC 相对地址加载)，使得常量可以嵌到代码段中（典型做法就是放在函数的开头之前）。这些是仅有的没有严格对应于正常的 MIPS 指令的 MIPS16 指令——MIPS 没有相对 PC 寻址的数据操作。

许多 MIPS 的 load/store 操作是直接在栈帧(stack frame)里，**$29/sp** 可能是最常用的基址寄存器。MIPS16 定义了一组隐式使用 **sp** 的指令，允许我们把函数的栈帧引用地址也编码进去而不需要一个单独的寄存器域。

MIPS 的 load 指令总是生成完整的 32 位地址。由于字加载指令仅当地址是 4 的倍数时才合法，最低两位地址就被浪费了。MIPS16 的 load 指令支持变址索引：地址的偏移量会根据待存取的对象的大小向左移位，从而增加了指令中可用的地址范围。

作为一种额外的应急机制，MIPS16 定义了一些指令，允许在 8 个 MIPS16 可访问的寄存器与 32 个 MIPS 通用寄存器中的任何一个之间做不受限制的数据传送。

B.1.2 对 MIPS16 ASE 的评价

MIPS16 不是一种适合汇编语言编程的语言，我们这里也不准备对它详细说明。它是给编译器用的。大多数使用 MIPS16 模式编译的程序的尺寸都会缩小到用 MIPS 模式编译的 60-70%。这比 32 位 CISC 体系结构的代码更为紧凑，跟 ARM 的 Thumb 代码差不多，和纯 16 位 CPU 相比有一定竞争力。

但是没有免费的午餐；MIPS16 程序可能比 MIPS 编译时多出 40-50% 的指令。这意味着在 CPU 核上运行一个程序会多用 40-50% 的时钟周期。但是低端 CPU 经常主要受存储器的限制，而不是受 CPU 核的限制。较小的 MIPS16 程序

需要较低的带宽来取指令，从而降低了高速缓存缺失率。在高速缓存很小并且程序的存储器有限时，MIPS16 将会缩小速度差距，还有可能赶上正常的 MIPS 代码。

由于性能的损失，MIPS16 代码在有大量存储器资源和总线带宽的计算机中没有吸引力。这就是为什么它只是一种可选扩展的原因。

在其应用范围的高端, MIPS16 将会与软件压缩技术展开竞争。一个普通的 MIPS 程序用通常的文件压缩算法压缩进 ROM 存储器之后，将会比未压缩的同等 MIPS16 代码小，而比压缩过的同等 MIPS16 代码略大；[2] 如果你的系统拥有足够的内存能够把 ROM 当做文件系统使用，而把代码解压缩到 RAM 中执行，那么用软件解压完整的 ISA 很可能会带来更好的总体性能。

也有这样一种明确的趋势，那就是大量使用以字节编码的解释语言（Java 或它的后续者）来书写在时间上要求不严格的大块程序。那种中间代码尺寸非常小，比任何二进制机器码都高效得多。如果只有解释器和一些对性能要求严格的程序采用机器自身的 ISA 书写，那么采用更紧凑的指令集编码格式将只会影响程序的一小部分。当然解释器（特别是 Java）本身可能会非常大，但是应用软件复杂度的无情增长将使解释器自身大小的重要性急剧降低。

我预料 MIPS16 将会继续在功耗、尺寸和成本受限制的系统中小范围应用。它还是值得发明和维护的，因为这当中有一些系统可能会批量生产。

B.2 MIPS DSP ASE

MIPS DSP ASE 的目标是克服传统的指令集在面对多媒体应用时所感到的不足。软件调制解调器的音频信号编码/解码、流媒体处理或者图像/视频压缩/解压等这些工作因为使用了基于复杂数学运算的算法，一度被认为属于专门的数字信号处理器的保留地。在计算层次上，这种多媒体任务常常涉及到对大批的向量或数组数据重复进行同样的操作。

在基于寄存器的机器内部，加速向量运算的一个好方法是把多个数据项打包到单个的机器寄存器中，用一条寄存器-到-寄存器的指令在每个寄存器的每个域上执行同样的运算。这是一种极为直接的并行处理，叫做 SIMD（代表“单指令多数据(single instruction, multiple data)”）。DSP ASE 包括四字节和双半字的 SIMD 操作。

不仅仅是向量操作，DSP ASE 还有几个别的特征：

- 定点小数类型：在这些环境中用浮点计算（从芯片面积和功耗角度来说）还不够经济。DSP 应用软件采用定点小数。这样一个小数只是一个带符号的整数，但是解释成该整数除以 2 的某次方所得的值。隐含的相除因子是 2^{16}(65536) 的32 位的小数格式称作 Q15.16 格式；那是因为有 16 位用作小数部分的精度，15 位用作整数部分的范围（最高位用作符号位不算在内）。

[2] 更紧凑的编码格式让压缩算法可利用的冗余度降低。

采用这个记法，Q31.0 就是传统的带符号整数，Q0.31 是介于 -1 和 1(接近1) 之间的小数。结果表明 Q0.31 是 DSP 应用中最受欢迎的 32 位格式，因为相乘时不会溢出（除了极端情况 -1×-1 导致太大的值 1）。Q0.31 常简写为 Q31。

DSP ASE 提供对 Q31 和 Q15（有符号 16 位）小数的支持。

- 饱和算术运算：对 DSP 算法内建溢出检查是不现实的——它们要求速度很快。可以设计一些巧妙的算法以避免溢出；但是并不是总能有这样的算法。在计算结果溢出时通常危害最小的做法就是把结果用可以表示的最大或者最小的数代替。这样执行的算术操作称为饱和 ——DSP ASE 中的很多运算都实现了饱和特性（在许多情况下同一个指令同时有饱和与非饱和两个版本）。

- 小数乘法：如果你通过复用全精度的整数乘法器把两个 Q31 小数相乘，那么你得到一个 64 位结果，由一个 Q62 结果和（在最高位上）复制的符号位构成。这个结果表示方式比较古怪，把这个值左移一位生成一个 Q63 的格式（对 64 位的使用更为自然）更有用些。生成 Q31 值的 Q15 乘法也要左移。

- 舍入：有些小数操作默认丢弃最低有效位。但是如果被舍弃的位表示的值超过新的最低位为 1 的值的一半时，将结果进一位近似结果更好。这就是 DSP ASE 里舍入的含义，类似十进制的四舍五入。

- 乘累加操作有四个累加器可选：(用于定点类型时可能会饱和)。

  象 MIPS 公司的 24-K CPU 这种现代的 MIPS32 CPU 已经有一个相当成熟的整数乘法累加运算，但是用于小数和饱和运算时效率并没有那么高。

  有了四个 64 位结果/累加器寄存器 **ac0–3** 以后，乘法累加操作更好用。老的 MIPS 乘除单元只有一个累加器，通过 **hi/lo** 寄存器访问。新的 **ac0** 就是老的 **hi/lo**，不过起了个新名字。

DSP 指令集一点都不象常规的正交的 MIPS32 指令集。它是一些用于特殊目的的指令的集合，很多时候针对的是重要算法中已知的热点。

有关 DSP 指令的一览表，参见 MIPS 公司出版的 34-K 和 24-K CPU 的程序员手册。

B.3 MDMX ASE

MDMX 是一个要比 DSP ASE 老得多的尝试，在 MIPS 指令集的范围内提供类似 DSP 的 SIMD 多媒体运算。MDMX 是 SGI 开发和倡导的，于 1997 年公布。

二者采用的方法存在着巨大的不同。MDMX 选择了在浮点单元的 64 位寄存器里处理多媒体数据，而 DSP ASE 采用通用寄存器，多数是作为整数乘法单元的扩展而构建的。MDMX 的选择可能更适合于讲究最高性能或者编程方便性的场合，但是要占用更多的硅片面积。

DSP ASE 的引入为我们质疑 MDMX 在实际软件中得到广泛应用的可能性提供了又一个理由，所以在此我们不多谈 MDMX。如果本书还会有第三版的话，也许那时候我们会发现这样做其实是一个错误而不得不纠正。

MIPS 术语表

$f0–$f31 寄存器： 32 个通用 32 位浮点寄存器。在 MIPS I(32 位) CPU，只有偶数编号的寄存器可以用作算术运算（奇数号寄存器保存 64 位双精度数的低 32 位）。

$n 寄存器： CPU 的 32 个通用寄存器之一。

a0–a3 寄存器： CPU 寄存器 $4–$7 的别名。

ABI(Application Binary Interface): 构建程序二进制代码的特定标准，人们认为在符合该标准的环境下能够保证正确执行。MIPS32 CPU 通常用一个众所周知的 o32 标准，但 64 位操作，可以选择 n32 和 n64。许多嵌入式系统依赖于整个 API 的一个小的子集。

累加器（accummulator）: 特别指定的寄存器来捕捉重复加减运算的结果。在 MIPS 体系结构中，乘法单元结果寄存器 hi/lo 构成累加器

地址区(address region): 指的是 MIPS 的程序地址空间划分为 kuseg、kseg0、kseg1 和 kseg2。参见每个单独区域名字下面的内容。

地址空间(address space): 应用程序可见的整个地质范围。在保护的操作系统下运行的程序要检查地址的有效性并进行转换。因为这样一个操作系统可以同时运行许多应用程序，有许多地址空间。

Alchemy Semiconductor: 1999 年起来的公司，由主要的 StrongARM 设计人员离开 DEC，意在制造低功耗、中等性能、高度集成的 MIPS 微处理器。他们把成功寄托在“袖珍计算机”的爆炸式增长的预言之上，但是这一预言并未成为现实。2002 年 Alchemy 成为 AMD 的一部分，2006 年成为 Raza 的一部分。其 AU1xxx 产品线在向嵌入式系统供货。

Algorithmics: 一家英国公司，专门从事 MIPS 技术和工具，我是其合伙人之一。

对齐(alignment): 数据在存储器中定位到特定的字节边界。如果数据开始的地址除以自身所占空间大小所得余数为零，则称为自然对齐。MIPS

CPU 要求其机器支持的数据对象自然对齐；因而字（四字节）要对齐到四字节边界，浮点 double 数据必须对齐到八字节边界。

alloca: C 库函数，返回一块在调用该库函数的函数返回时自动回收的存储区域。

Alpha: 由 Digital Semiconductor 公司制造的 RISC CPU 类型；算是 MIPS 近亲。

算术逻辑单元 ALU(arithmetic/logic unit): 用来指 CPU 中执行运算功能的单元的术语。

AMD: 一家存在很久的微电子公司，在 2004 年前后因为造出了比 Intel 公司更好的 x86 CPU 而出名。本书提到它是因为它收购了 Alchemy Semiconductor 后有了一个 MIPS 的嵌入式产品线。

分析仪(analyzer): 参见逻辑分析仪(logic analyser)。

Apache 小组(SVR4.2): 在 UNIX System V Release 4.2 操作系统和 *MIPS ABI* 标准结合的基础上，提供 MIPS 体系结构 UNIX 系统的供应商的联盟。

应用程序接口 API(Application Program Interface): 对某些软件功能的过程调用接口，通常以一组标准化的 C/C++ 函数调用形式提供。这是一个伟大的想法，往往被获得清晰接口后面有用的行为及其困难这一事实打乱了。

体系结构(architecture): 参见指令集体系结构*(instruction set architecture)*。

档案(archive): 目标代码库的另外一种叫法。

参数(argument): C 语言传递给函数的值。别的语言往往称作参量(parmater)。可以理解为 C 的参数是传值调用的参量。

ARM: 总部设在英国的一家发放 ARM 体系结构的处理器核的授权的公司。ARM 体系结构基本上是 RISC，但其最初设计是在流水线的方便性和电路的简单性之间更追求方便性。然而其简单性帮助 ARM 公司成为了第一个有竞争力的 32 位软核 CPU 的供应者。

ASCII: 非常松散的用于 C 语言的字符编码。

ASE: MIPS 体系结构的专用扩展。在 MIPS32/64 体系结构基础上定义了好几个可选的扩展。ASE 的存在用 MIPS32/64 定义的一个标志表示，通常是某个 Config CP0 寄存器中的域。

专用集成电路 ASIC(application specific integrated circuit): 为了用于某个特定电路专门设计或者调整的芯片。

ASIC-core CPU: 参见 CPU core。

ASID: 在 CPU 的 EntryHi 寄存器中维护的地址空间 ID。用来选择一组特定的地址转换：只有那些自身的 ASID 值和当前值匹配的地址转换才会给出有效的物理地址。

汇编器，汇编代码(assmble, assembly code): 汇编代码（有时称为汇编语言程序或代码）是可供人阅读的机器指令的形式。汇编器就是把汇编语言程序翻译成可执行程序的程序，可能要经过一个中间的目标代码。

相联存储(associative store): 可以用给出存储数据的一部分进行查找的存储器。对于每个数据域需要一个单独的比较器，所以大的相联存储器用掉大量的逻辑。MIPS TLB 如果安装了的话，采用一个 32 到 64 项的全相联存储器。

相联度(associativity): 参见 *cache, set-associativity*。

异步逻辑(asynchronous logic): 异步电路就是不是围绕着一个或几个时钟信号组织的电路。参见同步逻辑*(synchronous logic)*。

(ATMizer): 由 LSI 为 ATM 网络接口制造的部件；有一个内置的 MIPS CPU 作为其中一个部件。

原子的、原子地、原子性(atomic, atomically, atomicity): 用计算机科学的行话来讲，如果一组操作要么全部都执行要么全部都不执行就称为原子的。

回溯(backtrace): 参见栈回溯*(stack backtrace)*。

无效虚拟地址寄存器(BadVAddr register): 用来保存因为某种原因（未对齐、不可访问、TLB 未命中等等）导致自陷的地址值的 CPU 控制寄存器。

防护、遇险(barrier, hazard): 参见遇险防护*(hazard barrier)*。

防护、存储器(barrier, memory): 参见 sync 指令词条。

BAT: 在没有存储器映射硬件（TLB）的 MIPS CPU 中复用程序存储区的一个（基本上过时的）选项。

bcopy: C 语言用来拷贝一块存储器内容的库函数。

基准测试(benchmark): 可以运行于许多不同计算机上用以比较相对性能的程序。基准测试从最造的用来测量特定任务的代码片断演化成为大量套件，这样对于一个系统处理常见应用程序的速度可以提供指导信息。

Berkeley Unix: 参见 BSD Unix。

引导异常向量 BEV(boot exception vectors): CPU 状态寄存器中的一位，可以导致自陷经过位于不进行高速缓存的(kseg1)存储区的一对备用的向量。该位置接近复位时的起点这样两个都可方便的映射到同一只读存储器。

bias: 参见 floating-point bias。

BiCMOS: 一种特殊的芯片制造技术，混合采用高密度低功耗的 CMOS 晶体管用于内部逻辑，快速和翻转次数少的双极性晶体管作为接口。在 1980 年代后期对 CPU 很时髦，但是没有人用这种技术取得太大成功。

大尾端(big endian): 描述一种把多字节整数的最高有效位字节存于最低字节地址的体系结构；参阅第 10.2 节。

位域(bit field): 字的一部分，解释为单个位的集合。

块大小(block size): 参见 *cache line size*。

阻塞(blocking): 一种在完成之前停止执行的操作。

自举(bootstrap): 负责从一种 CPU 或系统处于不确定的状态下启动的程序或者程序片断。

分支(branch): 在 MIPS 指令集里是相对于 PC 的转移。

分支并链接(branch and link): 相对于 PC 的子程序调用。

分支延迟槽(branch delay slot): 位于存储器中的指令序列中紧跟一个转移/分支指令后面的位置。处于分支延迟槽中的指令总是在分支目标指令之前执行。有时候有必要用一条 nop 指令填充分支延迟槽。

分支优化(branch optimization): （由编译器、汇编器或者程序员进行的）为了充分利用分支延迟槽而对内存中的指令次序进行调整的过程。

分支开销(branch penalty): 许多 CPU 在分支后停顿几个时钟周期，因为已经将分支之后的指令取进了流水线所以必须回溯并重新填充流水线。这个延迟（以时钟周期为单位）称为分支开销。在短流水线上为零，但是在长流水线的 R4000 两个时钟周期的分支开销对效率影响很大。

分支预测(branch predicaion): 一种 CPU 实现技术，其中工作于流水线前端的简单逻辑识别分支指令并且猜测分之目标。流水线的取指阶段根据猜测进行，而不是一直顺序取指。CPU 需要某些机制在发现猜测错误时进行回退。

结果表明不需要太高复杂度的分支预测效果就很好，除了最简单的 32 位 CPU 之外普遍都采用。

BrCond3–0: 由协处理器条件分支指令直接测试的 CPU 输入信号。

断点(breakpoint): 当调试程序的时候，断点就是调试生成自陷并将控制返回用户的指令位置。通过粘贴一条 break 指令到被调试程序中实现。

Broadcom Corporation: 一家知名的计算机网络芯片制造商，这里提到是因为它拥有和生产 SB12xx 系列 64 位 MIPS CPU，集成到用途很广的许多网络接口设备中。SB-12xx 的设计来源于 SiByte，见下文。

BSD UNIX: 伯克利软件发行（Berkeley Software Distribution）是最初运行于 DEC VAX 小型机上的 UNIX 操作系统的变种。BSD 是第一个拥有完全的 32 位虚拟存储内核的 UNIX 变种，成为后来 Sun 最早的操作系统的基础。经过很长时间的争论，BSD 4.4 版以开放源码方式发布（但是该争议持续了很久激起了 Linux 的开发成为一个后继的内核）。它最新的直接的后代就是苹果公司的 OS X。

BSS: 在编译后的 C 程序中，用来容纳没有明确初始化的声明的变量的存储区。对应于目标代码的一个区。好像没有人能够记得 BSS 代表什么意思。

大多数 C 编译系统用 ".bss"名作为分配了空间但没有赋初值的全局变量数据区。

突发读周期(burst read cycle): MIPS CPU （除了很早期的一部分之外）在一个突发读周期从内存一次读取多个字（常见为四个字）来填充高速缓存。

字节(byte): 8 位的数据。

字节序(byte order): 用来强调数据在内存中按照字节地址的次序。看上去好像很明显，但是当考虑组成字和半字的各个字节时容易让人混淆。

字节交换(byte-swap): 反转组成一个字的各个字节的次序的操作。这在对来自不同尾端的机器的数据进行调整的时候要用到。

C 预处理器(C preprocessor): cpp 程序，一般用作 C 编译器的第一遍。cpp 负责文本替换和省略。它处理注释和以"#"开始的有用的预处

理指令，例如 `#define`、`#include` 和 `#ifdef` 等。尽管和 C 相关联，但它是一个通用的宏语言，可以也常常用于其它语言。在本书中，重要的非 C 的应用是用于对汇编语言程序进行预处理。

C++: 一种编译型语言，保留了大部分 C 的语法和外表，但是提供了一整套面向对象的扩展。

高速缓存(cache): 一个小容量的辅助存储器，距离 CPU 很近，复制最近从内存读取的数据内容。在第 4 章全面讲述 MIPS 的高速缓存。

高速缓存，直接映射(cache, direct-mapped): 直接映射的高速缓存，对每个具体的存储器位置，只有一个槽可以存放该处的内容。如果一个程序碰巧频繁使用恰好要用一个高速缓存槽的不同内存位置处的两个变量，直接映射的高速缓存特别容易变的低效；但是直接映射的高速缓存简单所以可以运行于更高的时钟频率。也许更重要的是，直接映射的高速缓存需要较少的布线连接控制器和存储单元，这使得直接映射的高速缓存成为片外高速缓存的自然选择。

即使在最简单的 CPU 里，直接映射的高速缓存现在也很少见了。

高速缓存，重复标签(cache, duplicate tags): 在高速缓存一致的多处理器中，总线接口控制器必须经常察看 CPU 的高速缓存—具体说，是察看特定高速缓存标签—以检查特定的总线事务是否和高速缓存当前的数据有关。这种访问成本很高，如果总线接口和 CPU 分时使用标签，那么 CPU 的延迟成本很高，如果是双端口的标签时，那么硬件和互锁的成本很高。更节省的做法通常是对总线接口感兴趣的高速缓存信息保留第二份拷贝，和主高速缓存并行更新—反正导致任何一个变化的事件都是总线可见的。重复标签并非要完美才能有用；只要它能让总线接口在多数情况下避免访问 CPU 的标签，就仍然能够提高系统的效率。

高速缓存，物理寻址(cache, physical-addressed): 完全采用（转换后的）物理地址进行访问的高速缓存。早期 MIPS CPU 的高速缓存以及所有的 MIPS L2 高速缓存都是如此。

高速缓存，组相联(cache, set-associative): 对来自一个具体存储器位置的数据有多个地方可以存储的高速缓存。常见的是两路组相联高速缓存，就是说，对于存储器内任意的数据都有两个可用的高速缓存槽。等效于有两个高速缓存同时进行搜索，这样系统可以处理两个频繁访问的数据位于同一个高速缓存索引的情况。

一个组相联高速缓存需要比直接映射高速缓存更宽的总线，而且运行的没那么快。早期的 RISC 用直接映射的高速缓存以节省外部高速缓存的引脚。尽管宽总线对于片上高速缓存不成问题，有些早期集成的 CPU 仍然

有直接映射的高速缓存来提高时钟频率。现在，一般倾向于采用组相联的片上高速缓存因为有较高的命中率。

高速缓存，监听(cache, snooping): 在高速缓存中，侦听指监控某些别的设备（另外的 CPU 或者 DMA 主控器）的总线活动以查找对位于高速缓存的数据的引用。一开始，“侦听”这个词用来指可以干预的高速缓存，在高速缓存比存储器数据还新的情况下提供高速缓存的版本否则别的主控设备可能得到内存中老的数据；这个词现在用来指任何对总线活动进行监控的高速缓存。

高速缓存，分开的(cache, split): 这样一种高速缓存，对取指和数据访问采用分开的高速缓存。

高速缓存，回写(cache, write-back): 一种 CPU 在其中写入时数据保存在高速缓存中，但是（当时）并不送到主存的数据高速缓存(D-cache)。该高速缓存行标记为“脏(dirty)。” 当其它位置的数据需要用到该高速缓存行，或者针对该高速缓存行执行一次回写操作时，数据才被写到主存。

高速缓存，透写(cache，write-through): 一种每次写操作都同时写入高速缓存（如果访问命中高速缓存）和主存的数据高速缓存(D-cache)。其优点是高速缓存从来不会包含没有写入主存的数据，所以可以随意丢弃高速缓存行。

通常，当存储器系统的（相对较慢的）写周期正在运行时，写往主存的数据可以存储于一个写缓冲器中，所以 CPU 不必等待写操作完成。

只要存储器周期快到能够吸收以略高于 CPU 的平均写操作产生率的速度到来的写操作，透写高速缓存就工作得很好。在现代的系统中很少见了。

高速缓存重影(cache aliases): 在虚拟存储的操作系统中，你有时可以把同样的数据映射到不同的位置。这可能发生在，两个任务不同的地址空间之间共享的数据，或者在应用程序和内核之间有不同角度的数据上。

现在许多 MIPS CPU 采用程序（虚拟）地址作为高速缓存索引—能够和地址转换并行开始高速缓存搜索因而可以节省时间。但是如果不同的程序地址可以访问同一数据，我们可能会碰到同一数据处于高速缓存中两个不同位置—即高速缓存重影。如果我们开始写入这些位置，就会发生严重错误。

高速缓存重影其实可以避免。MIPS CPU 采用的页地址转换意味着低 12 位地址在转换时保持不变，可能的最大高速缓存的索引也只用大约低 15 位。内核软件在对一个页生成多个不同地址的时候，只要小心保证分配的程序地址第 12–15 位相同就行了。

高速缓存一致性(cache coherency): 高速缓存总是精确给出你的程序或者系统的其它部分存储进高速缓存/内存子系统的内容的良好状态。为了追求一致性采用了许多复杂的技术和硬件窍门；用于“服务器”的老式 MIPS CPU(比如 R4000SC 和 R10000)在高速缓存中采用了特殊的机制来达到这一目标。但是这种技术在大型服务器计算机之外的世界里用得还不多。

高速缓存清洗(cache flush): 一个有些歧义的术语；最好避免使用。从来都不清楚它到底是指回写、作废还是二者的结合。

高速缓存命中(cache hit): 即当你在高速缓存中查找并且找到了所要的东西。

高速缓存索引(cache index): 高速缓存索引指用来从每组选择高速缓存位置的地址部分。

高速缓存作废(cache invalidation): 标记某一高速缓存行的数据不再使用。高速缓存行的控制位中总有某种有效位用于这个目的。这对 MIPS CPU 初始化是必不可少的一部分。

高速缓存行大小(cache line size): 每个高速缓存槽可以容纳一个或多个字的数据，用单个地址标签标识的一个数据块称为一个高速缓存行。大的行节省了标签所用空间可以提高重填效率，但是大的行因为加载了更多不需要的数据而浪费了空间。

最佳的高速缓存行宽倾向于随着离开 CPU 距离越远和较大的未命中代价而增加。MIPS I CPU 总是只有一个字的高速缓存行，但是后来的 CPU 倾向于采用四或八个字的行。

高速缓存未命中(cache miss): 指当你在高速缓存中查找但是没有找到要找的东西的情况。

高速缓存未命中代价(cache miss penalty): 当未命中高速缓存时 CPU 停顿所花费的时间，取决于系统的存储器响应时间。

高速缓存剖析(cache profiling): 测量一个具体的程序运行时所生成的高速缓存流量，目标在于重新组织存储器中的程序以减小高速缓存未命中的次数。除了极小的程序和程序段外，还不清楚这种做法的可行性有多大。

高速缓存重填(cache refill): 在一次高速缓存未命中后用来获取一个高速缓存行数据的存储器读取操作。这是第一次读进高速缓存，CPU 然后重新开始执行，这次就命中了高速缓存。

高速缓存组(cache set): 组相联高速缓存的一个块。

高速缓存模拟器(cache simulator): 用于高速缓存剖析的软件工具。

高速缓存标签(cache tag): 高速缓存行中存储的用来识别该数据在主存中位置的信息。

高速缓存回写(cache write-back): 把一个高速缓存行的内容复制回主存相应的存储块的过程。通常是条件执行的，因为高速缓存行有个“脏(dirtry)” 标志，用来记住自从上次从主存中取出以来是否被写过。

可高速缓存的(cacheable): 用于一个地址区域或者存储器转换系统定义的一个页。

高速缓存错误寄存器(CacheErr register): R4000 及其后代的 CPU 控制（协处理器 0）寄存器，充满了用以分析和修复高速缓存奇偶校验/ECC 错误的信息。

高速缓存操作(cacheop): MIPS 体系结构提供了一个多用途的 cache 指令用来初始化和操作高速缓存内容。

被调用者(callee): 在一次函数调用中被调用的函数。

调用者(caller): 在一次函数调用中，出现调用指令以及调用结束后控制返回到的函数。

原因寄存器(Cause Register): CPU 控制寄存器，在自陷后告诉你是那种类型的自陷。Cause 也给出哪个外部中断信号正在活动。

Cavium Networks: Cavium 的网络专用计算机芯片，2006 年发布，叫做 Octeon。每个芯片含有多个 MIPS64 CPU。

上界(ceiling): 浮点数到整数的转换，舍入到不小于该浮点数的最小整数。由 MIPS 指令 ceil 实现。

字符(char): 用来容纳字符编码的小整数在 C 语言中的名字。在 MIPS CPU（现在所有实际的 CPU）里，这是一个字节。

(CISC): 用来指非 RISC 体系结构的首字母缩写。在本书中，我们指象 DEC VAX、Motorola 680x0、Intel x86(32 位版本)之类的体系结构。所有这些指令集都是在 RISC 发现之前发明的，所有的都很难执行得比 RISC CPU 快。

时钟周期(clock cycle): CPU 时钟信号的周期。对于 RISC CPU，这是连续的流水线阶段运行的速率。

互补对称金属氧化物半导体(CMOS): 用来制造所有实际 MIPS CPU 的晶体管技术。CMOS 芯片比其它技术密度高功耗小，所以具有领先优势的集成。在 CPU 里，能够将很多电路放进更小的空间的能力被证明是关键的性能因素，因而全部快速的 CPU 现在都是 CMOS。

一致性(coherency): 见高速缓存一致性*(cache coherency)*。

比较寄存器(Compare reigister): 为实现 CPU 上的定时器而提供的 CPU 控制寄存器。有些很老的 MIPS CPU 并不提供该寄存器，但是你也许再也不会碰到这种 CPU 了。

配置寄存器(Config register): 在 MIPS32/64 CPU 上标准化的、用来配置基本 CPU 行为的 CPU 控制寄存器。

控制台(console): 被公认的向用户发送消息和接收用户输入的 I/O 通道。

常量(const): C 语言声明的属性，表示该数据只读。常常和指令打包到一起。

上下文寄存器(Context register): 只在带有 TLB 的 CPU 上有的 CPU 控制寄存器。提供了一种用某种页表安排在系统上处理 TLB 未命中的快速方法。

上下文切换(context switch): 多任务操作系统中从一个任务到另一个任务的软件环境变换的工作。

协处理器(coprocessor): CPU 的一部分，或者某些别的紧密耦合的机器部件，执行一些特殊的保留的指令编码的指令集。这是一个 MIPS 体系结构的概念，成功的分离了可选的、或者是具体实现相关的指令集部分因而减少了主流指令集的变化。这种做法相当成功，但是其命名导致了许多误解。

协处理器条件(coprocessor condition): MIPS 体系结构中每个协处理器子集的特殊指令都有一个单独的位表示和主整数 CPU 的通信状态，用 bcxt/bcxf 指令测试该状态。见第 3 章。

协处理器条件分支(coprocessor conditional branches): MIPS 体系结构的协处理器可以提供一个或多个条件位，所有协处理器都用 bclt label 根据条件的意义进行分支。

协处理器 0 (coprocessor 0): 与存储器映射、异常处理等类似的特权控制指令相关的 CPU 功能（定义相当模糊的）位。

CPU 核(core CPU): 作为 ASIC 的一个部件设计的微处理器，用来构成所谓的“片上系统。” MIPS CPU 越来越多的被用作核。

corExtend ASE: MIPS 公司某些 CPU 核上可选的指令集扩展，使得芯片设计者增加或使用新的、专用的计算指令相对容易点。

计数寄存器(Count register): 连续运行的定时器寄存器，在 R4000 之类的 CPU 和某些早期的 CPU 上有。

cpp: C 语言预处理程序。

CPU core: 参见 core CPU。

提取公共子表达式 CSE(common subexpression elimination): 可能是优化编译器中最重要的单个优化步骤，要求当编译器重复一个已经队同样数据做出的计算能够检测出来。然后可以考虑保存第一次的运算结果进行复用。这点尤其重要，因为其它的编译器变换可能导致重复的结果，用 CSE 进行整理要比在其它优化中构建类似的逻辑要简单得多。

周期(cycle): 时钟周期。

数据高速缓存(D-cache): 数据高速缓存（MIPS CPU 总是拥有单独的指令和数据高速缓存）。

(D-TLB): 有些 MIPS 处理器用很小的分开的地址转换高速缓存从主 TLB 填充的，以避免同时进行指令和数据地址转换的资源冲突。大多数 MIPS CPU 拥有指令端的 I-TLB，许多更快的 CPU 也有 D-TLB。其操作对软件是不可见的，除了偶尔会有从主 TLB 存取时花费的额外的时钟周期。

数据依赖(data dependencies): 在某个寄存器中生成一个值的指令和随后要用这个值的指令之间的关系。

数据通路交换器(data path swapper): 见字节交换器(byte-swapper)。

数据/指令高速缓存一致性(data/instruction cache coherency): 保持 I-cache 和 D-cache 一致性的工作。MIPS CPU 不做这项工作；当你写入或修改指令流的时候作废 I-cache 位置很重要。参见高速缓存一致性(cache coherency)。

调试模式(debug mode): 一种特殊的 CPU 状态，非常类似于异常模式，与 Debug(DM) 寄存器位相联系。发生条是异常时进入调试模式，当调试器程序运行一条 deret 时退出。参见第 12.1.4 节。

调试探针(debug probe): 一个连接开发软件的主机和 CPU 的 EJTAG 单元的小盒子，让你能够调试目标系统上的软件。这样的调试器不要求目标系统上建有 EJTAG 单元及接口之外的资源，所以可用在大多数裸体的嵌入式设备。

调试器(debugger): 控制和查询正在运行的程序的软件工具。

(DEC): Digital Equipment Corporation，整个 1980 年代最为成功的小型机制造商。其 PDP-11 和 VAX 计算机给了 UNIX 以灵感。从 1989 年后它曾经涉足 MIPS CPU，时间虽短但是影响不小。

(DECstation): DEC 为其于 1989 到 1993 年之间推出的采用 MIPS 体系结构的工作站起的名字。

延迟分支(delayed braches): 见 *branch delay slot*。

延迟加载(delayed loads): 见 *laod delay slot*。

按需调页(demand paging): 一个程序渐进加载的过程。依赖于一个操作系统和背后实现虚拟存储器的硬件—操作系统捕获对程序尚未加载进内存的部分的引用，读进相应的数据并作地址映射让程序看到的是该部分位于正确的地方，然后返回到程序，重新执行失败的引用。这称为分页，因为存储器转换单元和加载以称为页的固定大小的块为单位。

非规格化(denormalized): 当一个浮点数的值太小以至无法以通常通常精度表示时称为是非规格化。IEEE 754 定义的方式很难让硬件直接处理非规格化的表示，所以 MIPS CPU 被用来处理或者计算非规格数的时候总是自陷。

去引用(dereferencing): 一个花哨的名词，用来指顺着指针来获取所指向的存储器数据的操作。

诊断(diagnostics): “诊断测试”的简称，用来指那些软件，写出来不是为了完成具体工作，而是用来引发、检测和诊断计算机及附属硬件问题的。高度集成的芯片中加入了许多特性为诊断软件提供方便。

直接映射的(direct mapped): 见高速缓存，直接映射(cache, direct mapped)。

汇编命令(directive): 用来指汇编语言程序中不生成机器指令只是告诉汇编器要做什么的部分的一个术语—例如 .globl。也叫做伪指令（伪操作）。

脏(dirty): 在虚拟存储系统中，用来描述一个存储器页自从上次从二级存储器取出或者写回之后已经被修改的状态。脏页不能丢弃。

反汇编器(disassembler): 把存储器中的二进制指令序列转换成人可以阅读的汇编语言助记符列表形式的程序。

位移(displacement): 地址计算中用到的加到某个“基址”上的值。MIPS 常见的情形就是，几乎每次加载/存储地址都用一个寄存器中的基址加上一个编码进指令的 16 位有符号位移。

直接存储器访问 DMA(direct memory access): 一个外部的设备在没有 CPU 的干预下直接和存储器进行数据传输。

(double): C 和汇编语言中对于一个双精度（64 位）浮点数的名称。

双字(doubleword): MIPS 体系结构描述中用于指（不是用于浮点的）64 位数据的名字。

下载(download): 从主机向目标机传输数据的动作（如果有疑问，主机一般指用户连接的那台机器）。

(DRAM): 用来不准确的指大容量的存储器系统（通常是用 DRAM 部件建起来的）。有时用来讨论用 DRAM 构造的存储器的典型性质。

(dseg, drseg, dmseg): 虚拟存储区域（和 kseg2 区重叠），只有在调试模式才可以访问，用来为调试代码映射 EJTAG 调试单元资源

数字信号处理器 DSP(digital signal processor): 一种特殊风格的微处理器，瞄准定位于处理流数据，每个数据代表某种最终从模数转换器得来的取样值。DSP 专注于某些流行的模拟算法的速度：强调乘加运算的性能。与通用的处理器相比较，在运算精度、用高级语言编程的方便性、以及支持基本操作系统功能的特性等方面都有欠缺。但是 DSP 定义本身并不象 RISC 那么严格。

(DSP ASE): MIPS32/64 体系结构的一个扩展，增加了大量指令帮助处理多媒体应用；许多新的操作都是 SIMD、饱和型以及某种相乘累加。

重复标签(duplicate tags): 见 *cache, duplicate tags*。

双字(dword): MIPS 汇编语言给 64 为整数数据的名称，也叫 doubleword。

动态链接(dynamic linking): 用来指应用程序在运行时动态寻找和绑定子程序库的过程的名次，微软称为 DLL。大多数 Linux/MIPS 系统基于用户空间应用程序的动态链接。

动态变量(dynamic variables): 用来指名义上或实际上保存在栈上的变量（类似于 C 语言函数内部定义的变量）的名词，这是一种过时的叫法。

纠错码 ECC(error-correction code): 和数据存储在一起的校验位，不仅可以用来诊断错误而且允许用来纠正错误（假定错误只影响少数位）。有些 MIPS CPU 采用对高速缓存和内存总线上（也许还包括内存，不过那是系统设计时决定的）每个 64 位双字数据加上 8 个校验位的纠错码 ECC。

射极耦合逻辑ECL(emitter-coupled logic): 决定一个信号代表一还是零的一种电气标准。ECL 比更常见的标准 TTL 传输速度快、抗噪声能力强，但是代价是功耗高。现在基本上过时了。这个名字描述当初用于这种芯片的晶体管实现。

外部中断控制器 EIC(external interrupt controller): MIPS32/64 作为一个选项，向 MIPS CPU 提供一个不同的中断信令方法。它提供了多达 63 个直接不同的中断（对于嵌入式系统很有用，因为经常有大量的中断源），但是代价是要求系统设计人员用硬件布线实现中断优先级。参见 IPL，具体细节见第 5.8.5 节。

(EJTAG): MIPS32/64 CPU 的核内调试单元规范，软件可用，但是带有通过 JTAG（测试接口）引脚连接的调试探针用处更大。见第 12.1 节。

可执行的链接格式 ELF(executable and linking format): 为 UNIX 标准化制定的一个目标代码格式，是 MIPS ABI 的强制规范。

emacs: 文本编辑器里的瑞士军刀，真正的程序员必不可少的工具，emcs (吓人的)每次当你按下一个键时运行一个你选择的 Lisp 程序。几乎可以任意定制，不管做什么工作，你都可以从数不清的以前做过类似工作的人那里得到小而有价值的帮助。本书就是用它写的。

嵌入式(embedded): 描述作为一个（主要）不当做计算机的物体内部一部分的计算机系统。可以是从视频游戏机到玻璃炉控制器的任何东西。

仿真器(emulator): 见 *in-circuit emulator; software instruction emulators*。

尾端(endianness): 指一台机器是大尾端还是小尾端。见第 10 章。

endif: （`#endif`）cpp 里表示条件包含的代码段的结尾。参见 `#ifdef`。

EntryHi/EntryLo 寄存器(EntryHi/EntryLo register): 只在带有 TLB 的 CPU 中实现的 CPU 控制寄存器。用来和 TLB 倒腾数据。

异常程序计数器 EPC(exception program counter): 告诉你异常之后 CPU 从什么地方重新开始执行的 CPU 控制寄存器。

结尾部分(epilogue): 见*函数结尾部分(function epilogue)*。

可擦除可编程只读存储器 EPROM (erasable programmable read-only memory): 该设备最常用于提供系统引导的只读代码；这里非正式的用来指那些只读代码的位置。

errno: 大多数 C 函数库用来报告 I/O 错误的全局变量的名字。

ErrorEPC 寄存器(ErrorEPC register): R4x00 和后来的 CPU 检测高速缓存错误，为了能够在 CPU 处于某些关键的（但是常见的）异常处理程序执行到半路时也能检测错误，高速缓存检错系统提供了自己独立的寄存器来记住返回地址。见第 4.9.3 节。

异常类型码(ExcCode): Cause 寄存器中包含的表示刚发生的异常类型的位域。

异常(exception): 在 MIPS 体系结构中，一个异常可以是任何导致控制被转到两个自陷入口点之一的中断或自陷。

异常，IEEE (exception, IEEE): 参见 *IEEE 浮点异常（floating-point (IEEE) 异常）*。不幸的是，这是一种不同于 MIPS 异常的动物。

异常向量，异常入口点(exception vector, exception entry point): CPU 硬件在异常之后开始执行的（虚拟）存储器地址。“异常向量”其实用词不当，来源于可以用软件通过驻留内存的表来重新定义入口点的体系结构。在 MIPS CPU 里，异常处理程序入口点只能通过修改 EBase 寄存器中的单个基址—全体一致的—变动。

异常受害者(exception victim): 在异常时，受害者是指由于发生异常而（还）未能运行的第一条指令。对于由于 CPU 自身活动导致的异常，受害指令就是引发异常的指令。通常也是异常之后控制返回的地点；但是因为分支延迟的效果并非总是如此。

反常值(exceptional value): 超出正常表示范围的浮点值（无穷大、信号非数 NaN 等等）。如果软件允许，任何计算结果为这种值的时候都可以引发一个 IEEE 754 异常。

可执行文件(executable): 描述一个可以直接运行的目标代码。

指数(exponent): 浮点数的一部分，见第 7 章。

扩展浮点数(extended floating point): MIPS 硬件并不提供，通常指使用超过 64 位（通常为 80 位）空间的浮点数格式。

外部的(extern): C 语言中在另外模块中定义的变量的数据属性。

FastMath: 参见 *Intrinsity* 词条。

错误，出错(fault, faulting): 见页面错*(page fault)*。

浮点单元条件码FCC(floating-point unit condition code): MIPS I 到 MIPS III 只有一个，后来的 ISA 有八个。

FCSR 寄存器(FCSR register): 浮点控制/状态寄存器，见第 7 章。

先进先出 FIFO (first-in, first-out): 临时容纳数据的队列，其中每一项以和进入队列相同的顺序出来。

修正(fixup): 在二进制代码中，指链接器/定位器为了使得和程序最后运行的位置相匹配而调整指令和数据区地址的活动。

标志(flag): 这里（计算机书中常用）用来指控制寄存器中单个位构成的域。

浮点偏置(floating-point bias): 一个加到 IEEE 格式的浮点数的指数部分的偏移量，使得对所有合法值的指数部分都为正。

浮点条件码/标志(floating-point condition code/flag): 由浮点比较指令设置的单个位，传回主 CPU 用 bclt 和 bclf 指令测试。

浮点仿真自陷(floating-point emulation trap): 当 CPU 不能实现浮点（协处理器 1）操作时发生的自陷。可以构建一个软件自陷处理程序来模拟 FPU 的行为返回控制，这样应用程序软件不需要知道是否安装了 FPU 浮点硬件。软件例程可能要慢 50–300 倍。

浮点异常(floating-point exception): 浮点计算的 IEEE 754 标准考虑到了计算结果可能是“反常值”—一个称呼各种各样用户可能不高兴的结果的词。该标准要求实现的 CPU 允许捕获每一种类型的异常—这样就有些混乱，因为 MIPS 捕获一般性事件的机制也叫做异常。

浮点单元 FPU(floating-point unit): MIPS CPU 执行浮点数学运算部分的名称。历史上曾经是一块单独的芯片。

foo: 万能的垃圾或者无用文件名。

FORTRAN: 早期科学和数值计算所用的计算机语言，在这些领域其很好的可移植性胜过了各种令人讨厌的缺陷。

FP: 浮点。

fp 帧指针寄存器（fp(frame pointer) register）: CPU 通用寄存器（$30），有时用来表示一个栈帧（*stack frame*）的基址。

fpcond: 浮点条件位（也叫做协处理器 1 条件位）的另一个名字。

FPU: 浮点单元。

小数，小数部分(fraction, fractional part): 浮点值的一部分（也叫做尾数*(mantissa)*）。见第 7 章。

帧，帧大小(frame, framesize): 见栈帧*(stack frame)*。

自由软件基金会(Free Software Foundation): 自由软件的孤独的守望者。FSF 是一个持有 GNU 软件版权的（松散的）组织。

全相联(fully associative): 见相联存储*(associative store)*。

函数(function): C 语言对子程序的叫法，本书大多数时候用这个名称。

函数结尾部分(function epilogue): 在汇编语言中，在函数尾部见到的固定指令和命令序列，与控制返回到调用者有关。

函数内联(function inlining): 高级编译器提供的一种优化方法，其中函数调用被用被调用函数的整个函数体的全部指令序列进行替换。在许多体系结构中，这能取得较大的效果（对于很小的函数），因为消除了函数调用开销。在 MIPS 体系结构中函数调用的开销可以忽略，但是内联有时候仍然是有价值的，因为它允许优化器根据函数上下文环境进行工作。

函数首部(function prologue): 汇编语言中，位于函数开头部分的固定的指令和命令，保存寄存器和建立栈帧。

gcc: GNU C 编译器的通常名字。

gdb: GNU 调试器，GNU C 的伴随工具。

全局的(global): 程序员对于名字和值在整个程序范围内可以访问的数据项的老式叫法。有时不严格的推广到任何拥有自己存储空间的有名数据项—这个正确的叫法是静态的(static)。

全局指针(global pointer): MIPS gp 寄存器，在某些 MIPS 程序中用作对定义为位于固定程序地址的 static 或 extern 的 C 数据项提供高效访问。

globl: 汇编语言对从模块外部可见的数据或代码入口点属性的声明。

GNU: 自由软件基金会项目的名称，该项目提供一个类似于 UNIX 的操作系统的所有成分（内核可能除外）的一个可以自由发布的版本。

GNU C 编译器(GNU C Compiler): 在独立程序员和自由软件基金会精神领袖 Richard Stallman 和世界各地志愿者的非凡的集体协作下产出的自由软件产品。除非你在用 SGI 的工作站，否则 GNU C 是 MIPS 上最佳的编译器。

全局偏移量表GOT(global offset table): MIPS/Linux 用的动态链接机制必不可少的一部分，最初是在 MIPS/ABI 名义下开发的。见 16.2 节。

gp 寄存器(gp register): CPU 寄存器 $28，常用做程序数据的指针。可以链接到该指针值 $\pm 32K$ 范围内的程序数据可以用单个指令指令加载。并非所有工具链、所有运行时环境支持这一点。

半字(halfword): MIPS 体系结构对 16 位数据类型的称呼。

遇险(hazard): 也叫做*流水线遇险(pipeline hazard)*。遇险发生在指令序列不可靠的地方，因为生产者的指令事实上可能还没有施加到程序序列中后面的消费者指令上（常常是由于生产者的效果在流水线后期才生效）。

当 CPU 体系结构并不坚持要求硬件效果透明时才会有遇险：有可能定义一个没有遇险的体系结构，以实现的复杂度和（可能的）性能损失为代价。在最老的 MIPS CPU 中，有一个遇险在用户特权级程序可见：企图在紧接加载指令之后的指令中使用数据就会失败。现代的 MIPS CPU 在用户特权级代码中没有遇险。

但即使新的 CPU 也有与 CPU 控制相关的遇险，这些在第 3.4 节讨论过。

遇险防护(hazard barrier): 一条置于生产者和消费者之间的指令，用来保证消费者看到生产者的效果全部生效。

堆(heap): 在运行时分配的程序数据空间。

(Henrich, Joe): 《MIPS 用户手册》的令人尊敬的作者，几乎所有的官方的 MIPS ISA 都是从这本书派生而来。

(Hennessy, John): MIPS 的精神领袖和奠基者，Hennessy 教授领导了 Stanford 大学最早的 MIPS 研究项目。

(hi, lo): MIPS 整数乘法单元的结果寄存器。合在一起作为整数乘法累加操作的累加器。

命中(hit): 见*高速缓存命中(cache hit)*。

(I-cache): 指令高速缓存（MIPS CPU 总是有分开的指令和数据高速缓存）。当 CPU 读取指令时，查找 I-cache。

(ICE): 见 in-circuit emulator 词条。

(ICU): 中断控制单元

幂等的(idempotent): 一个数学名词，指执行两次和执行一次具有相同效果的操作（因而执行九次和 99 次等效）。搅拌你的咖啡是个幂等操作，但是加糖就不是。

当流水线的 CPU 发生异常时并随后返回到被中断的任务时，很难保证一切都恰好只执行了一次；如果能够让某些操作幂等，系统就不会受虚假的重复操作的影响。比如，MIPS 的所有分支指令都是幂等的。

IDT: Integrated Device Technology 公司，1980 以及 1990 年代 MIPS CPU 嵌入式应用的先行者。IDT 还在继续茁壮成长，但是 CPU 不再是其主要业务了。

(IEEE): 美国电子电器工程师协会(Institute of Electrical and Electronics Engineers) 的首字母缩写。这个行业团体制定了许多计算领域的标准。它的工作常常比其它标准化机构更加实用、合理和有建设性。

IEEE 754 浮点标准 (IEEE 754 floating point standard): 一个关于算术数值表示的工业标准。该标准强制规定了一组基本算术函数的精确行为，为开发可移植的数值算法提供了一个稳定的基础。

(ifdef, ifndef:): `#ifdef` 和 `#endif` 将 C 语言中的条件编译代码括起来。该特性实际上由 C 预处理器完成，所以也可用于其它语言。

立即数(immediate): 在指令集描述中，立即数值指嵌入到代码中的常数。在汇编语言中，指任意常数值。

实现 (implementation): 用作“体系结构” 的反面。在本书中，大多数时候指我们讲的在一个具体 CPU 怎样完成某件事情。

在线仿真器(in-circuit emulator): 一开始，指用能够精确模拟 CPU 行为并提供一些方式控制执行和检查 CPU 的模块取代 CPU 的设备。微处理器 ICE 单元不可避免地要基于某个微处理器（常常是较高级的）芯片的版本。

常常有可能不用 ICE 进行开发—ICE 太贵而且问题太多。

但是现代，这个词有时被用来指调试探头*(debug probe)*（见上），用不同的方法提供了相似的功能。

索引，高速缓存(index, cache): 见 *cache index*。

索引寄存器(index register): 用来定义哪个 TLB 数据项的内容将用来从 EntryHi/EntryLo 之间读写传送。

Indy: 由 SGI 在 1980 年代采用 R4000、R4600 和 R5000 64 位 MIPS CPU 制造的成本相对较低的工作站。

非精确的(inexact): 描述损失精度的浮点计算。注意这在大多数日常计算中发生的很多；比如 $\frac{1}{3}$ 就无法精确表示。IEEE 754 规定要求 MIPS CPU 可以对非精确结果自陷，但是没有人打开这种选项。

无穷大(infinity): 一个浮点数值表示太大（或太小）没法表示的值。IEEE 754 定义了正负无穷大的运算行为。

内联(inline, inlined, inlining): 见函数内联*(function inlining)*。

指令调度(instruction scheduling): 利用 CPU 的流水线前后移动指令以获取最大性能的过程。在简单的流水线化的 MIPS CPU 上，这通常就指充分利用延迟槽。这由编译器来完成，有时也由汇编器来做。

指令集体系结构 ISA(Instruction Set Architecture): CPU 的功能描述，精确的定义怎样处理任意合法的指令流（但是不必定义怎样实现）。MIPS32/64 是本书讲的主要的 ISA。

汇编指令合成(instruction synthesis by assembly): MIPS 指令集省略了许多有用和熟悉的操作（例如加载超出 $\pm 32KB$ 范围之外的常数的指令）。MIPS 体系结构的大多数汇编器接受用几条机器指令序列实现的指令（有时叫做宏指令）。

(int): C 语言整数数据类型的名字。该语言并没有定义用多少位来实现一个 `int`。这个自由度可以让编译器选择在目标机器上效率最高的实现。

互锁(interlock): 指一条指令被延迟到某个条件准备好才执行的硬件特性。在 MIPS 体系结构中有少数互锁。

中断(interrupt): 一个（若未屏蔽）可以导致异常的外部信号。

中断屏蔽 (interrupt mask): CPU 状态寄存器中每个中断一位的屏蔽，确定在给定时间允许哪个中断输入产生异常。

中断优先级(interrupt priority): 见 IPL。

可中断的(interruptible): 一般用来指一段中间可以容忍中断的程序（程序员因而允许发生中断）。

Intrinsity Semiconductor: Instrinsity 发布、制造和演示了 FastMath MIPS32 CPU（还有一个附加的阵列协处理器）主频高达 2 GHz —使其成为目前为止最快的 MIPS CPU。然而，该公司可能更专注于硅半导体设计技术而不是 MIPS CPU。

作废(invalidation): 见 *cache invalidation*。

中断优先级 IPL (interrupt priority level): 在许多体系结构上，中断输入具有内建的优先级；在同等或者高优先级的中断处理程序执行期间中断不会生效。历史上，MIPS CPU 没有做这些，把优先级留给了软件处理。

但是采用了可选的 EIC 中断系统的 MIPS32 CPU 在硬件中交流和实现中断优先级。见 5.8.5 节。

Irix: SGI 工作站/服务器上的操作系统（基于 Unix System V 之上的）。

ISA: 指令集体系结构。

中断服务例程 ISR(interrupt service routine): 中断异常调用的软件的另外一个名字。

发射，指令 (issue, instruction): 当讲到计算机实现的时候，发射指已经用了某些 CPU 资源开始做与某条指令相关的操作的时间点。

I-TLB: 一个微小的从 TLB 复制信息的硬件表，用来转换指令地址，这样不用和转换数据地址的硬件竞争。在早期的 MIPS CPU 里叫做“micro-TLB” 或“uTLB。” 对软件是不可见的，除非你自己对时间计数会注意到当取指要访问主 TLB 时执行会有一个时钟周期的停顿。

JPEG: 一个图像数据压缩的标准。

JTAG: 为了实现测试功能而连接电子设备的规格标准，可音译为“接探格”。JTAG 信号设计中的所有活动构件构成菊花链，允许全部东西共用单个访问点。实际情况从来并非完全如此，因为小的单功能芯片通常并不支持 JTAG；但它是电路板产品测试必不可少的一部分。

与本书比较有关的是，JTAG 信号提供一种连接片上 EJTAG 单元到调试探头*(debug probe)*的可行方式。

转移链接指令(jump and link instruction): MIPS 指令集对函数调用的名称，将返回地址（链接）置入 **ra** 寄存器。

k0 和 k1 寄存器(k0 and k1 registere): 两个约定为自陷处理程序保留的通用寄存器。很难设计一个完全不用寄存器的自陷处理程序。

内核(kernel): 最小的可以单独编译的操作系统单元，包含任务调度功能。有些操作系统（象 Linux）是整块的，有一个完成许多功能的大内核；有些是模块化的，有一个小内核周围加上些辅助任务。

核心特权级(kernel privilege): 对于保护的 CPU，指其中允许执行任何操作的状态。通常启动后处于该状态；在小系统或简单的操作系统中，一直处于该状态。

Kernighan, Brian: 公认的负责对 C 语言进行系统化的《C 程序设计手册（C Programming Handbook）》一书的作者之一（另一个是 Denis Ritchie）。任何程序员都不应该读有关 C 语言的其它书了。

杂凑(kludge): 工程师对快速临时到处是修补的解决办法的贬义的称呼。

kseg0, kseg1: 未映射的地址空间（其实也可以说是映射到物理地址低 512 MB 空间）。kseg0 用于使用高速缓存的引用而 kseg1 则用于不做高速缓存的引用。独立的程序，或者使用简单操作系统的程序很可能整个运行在 kseg0/kseg1 区。

KSU, KU: 状态寄存器（第 3.3 节）中核心/用户特权级域。

kuseg: MIPS 程序地址空间的低半部分，以用户特权级运行的程序可以访问该区域，并且所以地址都经过（CPU 装备的 TLB）转换。

L1、L2、L3 高速缓存(L1, L2, L3, cache): 一级、二级、三级高速缓存的别名。

滞后(latency): 由于某些单元或别的单元导致的延迟。存储器读取滞后是指存储器交付数据所花的时间，通常是比带宽更重要（但更容易被忽视）的参数。

叶函数(leaf function): 自身不包含别的函数调用的函数。这种函数可以直接返回到 ra 寄存器，典型情况下不用栈空间。

电平敏感(level sensitive): 信号（尤其是中断信号）的属性。MIPS 中断输入是电平敏感的；任何时间一旦激活并且未屏蔽就会导致中断。

库(library): 见*目标代码库(object code library)*。

行宽(line size): 见*高速缓存行宽(cache line size)*。

链接器、链接-加载器(linker, link-loader): 对分开编译的目标代码模块连成一块并对外部引用进行解析的程序。

Linux: 这是“GNU/Linux”操作系统的内核的名称。详见第 13 章。

小尾端(little-endian): 一种将多字节整数的最低有效部分存储在最低字节地址的体系结构；见 10.2 节。

ll, 连锁加载(ll, load-linked): MIPS32/64 的连锁加载、条件存储指令对多处理器和单处理器系统，都能提供一种有效的构建*原子操作*和其它基本操作系统控制单元的有效方法。见第 8.5.5 节。

LLAddr 寄存器(LLAddr register): R4000 和后来的 CPU 的 CPU 控制（协处理器 0）寄存器，在诊断软件之外基本没有用到。该寄存器保存上一次连锁加载(ll)指令的地址。

lo、hi 寄存器(lo, hi register): 整数乘除单元的专用输出寄存器。这些寄存器是互锁的—试图从其中向通用寄存器传输数据的操作会一直阻塞到乘除操作结束。

加载延迟槽 (load delay slot): 在最初的第一个商业化的 MIPS CPU 中，加载延迟槽不仅是个开销，而且是硬性规定：硬件对加载值不进行互锁，由软件负责保证在紧跟的指令中不要使用加载值。

紧跟加载之后的指令位置称为加载延迟槽，类比于分支延迟槽（当然后者一直是体系结构的一部分）。

编译器、汇编器或程序员可以移动代码尽量利用加载延迟槽，但在老式的 CPU 上，有时只能在那里放一条空操作。

加载/存储体系结构(load/store architecture): 用来描述象 MIPS 这类体系结构，存储器数据的访问只能明确通过加载和存储指令进行。许多别的体系结构定义了隐含访问存储器的指令（比如堆栈操作，对存储器变量运算等）。

加载到使用的开销，加载延迟(load-to-use penalty, load delay): 在大多数 MIPS CPU 中，加载数据的提供不可能早到能够让紧接其后的指令没有延迟就立即使用（即使是高速缓存命中的情况下，试图紧接着就访问加载数据也至少有一个指令的停顿）。

源于紧接着就使用加载值而导致延迟的 CPU 周期数称为加载-使用延迟。民间流传的说法是，加载-使用延迟一个周期无伤大雅，两个麻烦不断、三个不可救药。

加载器(loader): 为了生成可执行文件而对目标代码模块进行处理并给指令和数据分配程序地址的程序。

局部变量(local variable): 只在当前编译/汇编的模块内部可以访问的有名字的数据项。

引用的局部性(locality of reference): 指程序的大部分存储器访问集中于一小部分存储器位置（至少短期内如此）的倾向。这一点正是高速缓存有效的原因。

逻辑分析仪(logic analyzer): 同时监控许多信号的逻辑电平（即 0 或 1）的测试设备。常用来保存微处理器访问过的地址列表。

(long): C 的额外精度的整数；对于兼容 32 位 MIPS CPU 的程序，和 `int` 一样都为 32 位。对于 64 位 MIPS 程序，可能也为 64 位，但是通常和指针长度一样。

循环展开(loop unrolling): 高级编译器采用的一种优化方式。程序中的循环被编译成能够在不分支到代码外部实现好几步循环的代码。这在由于长流水线和指令预取使得分支成本相对较高的实现（MIPS 很少如此）上特别有优势。即使在 MIPS 体系结构上，循环展开也有优化效果。通过将不同的循环步骤代码混合，就有改进指令调度的机会。

最近最少使用 LRU(least recently used): 当维护高速缓存就是回收最近最少使用高速缓存空间时用的最优替换算法。对于大的分组维护 LRU 状态非常复杂，所以严格的 LRU 很少用于超过四项的分组。

LS: 最低有效 (least significant)。

LSI: LSI Logic Corporation，一家制造 MIPS CPU 的公司—如今主要是制造要集成到客户的片上系统中去的 ASIC 核的一部分的 MIPS CPU。

宏(macro): 计算机语言中的一个在编译/汇编之前要用事先定义的文本进行替换的“单词。” 更具体地说，是用 C 的预处理语句 `# define` 定义的东西。

madd: 见*乘加(multiply-add)*。

主干机(mainframe): 大型昂贵计算机的老名字；拥有大量存储器、用于数据密集型的应用。

尾数(mantissa): 浮点数表示的一部分。（也称为小数或小数部分）见第 7 章。

被映射的(mapped): 用来描述程序生成的那些需要经过复杂转换才能得到物理地址的程序地址范围的词。

掩码/屏蔽码(mask): 一个位域，通过逐位逻辑“与”操作选择数据结构的一部分。

MDMX ASE: 对 MIPS32/64 的一个扩展（该扩展出现的比 MIPS32/64 早）。MDMX 采用浮点寄存器来表示小的（8 位和 16 位）整型数组，并提供对数组中全部整数同时执行的同样算术或图形操作。有点类似于 x86 的 MMX 扩展。

普遍认为这种操作对于加速音频和视频处理（多媒体）的通用任务非常有用。但是这些天，你见到更多的可能是 *DSP ASE*，支持同样的操作，但只需要通用寄存器。

memcpy(): 标准 C 库函数中用来拷贝数据块的函数。

存储器防护(memory barrier): 见 *sync* 词条。

微 TLB (micro-TLB, uTLB): *I-TLB* 的老名字。

微代码 (microcode): 许多 CPU 由一个底层的可用广泛晦涩的机器语言（微代码）编程的底层的核（微引擎）构成。官方的 ISA 中的指令通过微代码的子程序实现。

在 1970 年代，微代码方法有利于管理 CPU 设计的复杂性。随着 1980 年代开发出更好的设计工具，尤其是开发出了更好的电路模拟器之后，回到直接在硬件上实现 ISA 操作变得有可能了。但是许多 CPU（特别是 CISC 型的）仍然采用微代码实现复杂或生僻的指令。

小型机 (minicomputer): 1970 到 1980 年代的“冰箱大小”的计算机。这种计算机在 1970 年代售价相对较低，在大学的单个系或者小生意的预算可承受的范围内。第一个 UNIX 和后来的开放源码运动就起源于小型机，尤其是 DEC 的 PDP-11。

MiniRISC: LSI 公司的一系列为小型设备优化的 MIPS CPU 的商标名。

Minix: 由 Andrew Tannenbaum 写的带有某些 UNIX 特征的简单小巧的操作系统。在入门的操作系统课程教学中所用的操作系统中最有影响力。值得一提的是，Linux 就是因为 Linux Torvalds 对于 Minix 仅用于教学的一些局限性不满意而写的。

MIPS: 在本书中，我们用这个词作为体系结构的名字。MIPS 是 MIPS 公司在美国和其它国家为其自己的体系结构、可综合的软核、硬件及软件所注册的一个商标名。

MIPS/ABI: MIPS 应用程序最新的标准，得到了采用大尾端形式的 MIPS 体系结构的全部 UNIX 系统厂商的支持。

MIPS 计算机系统有限公司(MIPS Computer Systems, Inc): 最初对 MIPS 体系结构进行商业化并销售的机构。有时候也用来指其后继者，即 SGI 公司的 MIPS 技术部门。

MIPS 半导体厂商(MIPS silicon vendor): 任何制造和销售 MIPS CPU 或含有最初设计的 MIPS CPU 组件的公司（即排除那些只用了别人的 CPU 核的公司）。这个名单包括：LSI、IDT、Performance Semiconductor、NEC、Siemens、Toshiba、NKK、Philips、QED、Sony、SGI、PMC-Sierra、SiByte/Broadcom、Alchemy/AMD、Cavium、Raza。没有其它哪个 CPU 体系结构可以有这么长的一个名单。

MIPS System VR3, RISC/OS: 这些都是用来指同一个操作系统，从 UNIX System V Release 3 派生出的一个操作系统。这些移植也是 Irix 的根源之一。

MIPS 技术有限公司(MIPS Technologies, Inc): 设计 MIPS CPU 核的公司，MIPS32 及 MIPS64 体系结构的作者，从 1999 年之后到现在的所有 MIPS 体系结构的监护人。MIPS 技术公司是 SGI 的 MIPS CPU 部门分离出来之后成立的一家公司。

经过短暂的尝试了高端的“硬核” MIPS64 设计之后，MIPS 公司集中于一系列可综合的核设计。现在这些核在嵌入式系统中得到了广泛的应用。

MIPS UMIPS 4.3BSD: MIPS 计算机公司第一个操作系统，是 Berkeley 的 BSD4.3 版本的 UNIX 系统的一个移植版本。它在历史上有重要意义，因为有些 MIPS 体系结构的特性就是专门针对 BSD 的需求而设计的。

MIPSEB, MIPSEL: 这是从大多数 MIPS 编译器工具链请求大尾端或小尾端输出时所用的词。

对齐不当的(misaligned): 即未对齐的 *(unaligned)*。

存储器管理单元 MMU(memory management unit): 在 MIPS 中提供的唯一的存储管理硬件就是 TLB，可以将高达 64 页程序地址转换成物理地址。

MS: 最高有效(most significant)（部分）。

乘加、乘累加(multiply-add, multiply-accumulate): 将两个数相乘然后执行一次加法求和的单条指令。如果有一个专用（可能双倍宽度）的累加器就是乘累加，如果最后结果可以保存到通用寄存器中就是乘加。

这类指令对于数值算法编码往往强大有效，尤其是对于浮点数。MIPS32 之前的 CPU 没有都实现整数乘加；实现该指令的基本上都兼容 MIPS32/64 的定义。

带有浮点硬件的 MIPS32/64 CPU 拥有完整的浮点乘加指令集。

DSP ASE 增加了一些专用的乘加指令。

多处理器(multiprocessor): 有多个处理单元的系统；实践中，我们把它用作 SMP 的同义词。

多任务(multitasking): 描述支持多个控制线程的操作系统。在最常见的层次上，线程用一个栈和一个下条指令的地址刻画，所以操作系统需要有某种调度器来挑选下次运行哪个线程，并保证所有任务都得到执行。

多线程硬件(multithreading hardware): 能够从硬件中的多个线程（同时，或至少在流水线中高度重叠）运行指令的 CPU。MIPS32/64 MT ASE 规定了 MIPS 实现多线程的一种方法，在附录 A 中有描述。

信号非数 NaN (Not a Number): IEEE 754 定义的特殊的浮点值，作为指令碰到非法的操作数时的返回值。

自然对齐(naturally aligned): 如果一个长度为 n 的数据的基地址模 n 为零，那么说这个数据是自然对齐的。一个字如果位于四字节边界，那么就说是自然对齐的；半字如果位于两字节边界就称为自然对齐；见对齐*(alignment)*。

NEC: 电子业的巨人，1990 年代领先的 MIPS CPU 的供应商。

嵌套异常/中断(nested exception/interrupt): 当你在执行上一个异常处理程序中间时再次发生 MIPS 异常时的情形。有时候嵌套异常没什么问题。

NKK: 作为一家大型的日本贸易公司的半导体部门，NKK 在 1990 年代早期生产了一些 MIPS CPU（主要作为 IDT 的第二个来源）。

不可屏蔽中断 NMI(nonmaskable interrupt): 在 R4000 及随后的器件中配备的（既作为一个输入信号也作为一个事件）。在 MIPS CPU 上，还不清楚到底是可屏蔽中断还是特定的软复位；二者没有真正差别。

noat, nomacro, noreorder: 汇编语言控制命令，提供给程序员一种方法来禁止某些并非总是用到的复杂的汇编器操作（没有 no 的命令名重新允许相应的操作）。

.set noat 防止汇编器将汇编代码翻译成用到 at/$1 寄存器的指令序列。

.set nomacro 防止汇编器将单个汇编语句翻译成多个指令。

.set noreorder 防止汇编器为了在分支延迟槽中填充有用指令而打乱代码次序。

非阻塞的(nonblocking): 一种操作，你可能期望它在完成之前停止执行，但是实际上避免了这样做。一条非阻塞的加载就是 CPU 越过加载指令继续执行，让数据加载在后台执行。加载数据肯定返回到一个寄存器中，未来指令对该寄存器的访问是互锁的，这样程序将在第一次使用加载数据时阻塞，除非此时已经执行到了。

非叶子函数(nonleaf function): 在某个地方调用别的函数的函数。正常情形下，编译器会生成一个函数首部，在栈上保存返回地址（可能还有别的寄存器值），以及一个函数尾部，恢复这些值。

非易失性存储器(nonvolatile memory): 适用于任何在系统断电之后仍然能保持数据的存储器技术。

空操作(nop, no-op): 没有操作。在 MIPS 上，nop 其实是指令 **sllv zero, zero, zero** 的别名，不产生什么效果，二进制的指令编码恰好为零。

规格化(normalize): 通过对尾数移位和对指数修改将浮点值转换成规格化的形式。除了很小的值之外，IEEE 的标准都是规格化的表示。

无效化(nullified): 适用于任何虽然已经发射并且继续进入流水线但却不许产生任何效果的指令—其写回寄存器的操作被抑制而且不得引发异常。在该指令属于"本不该执行"（也许是因为错误的猜测）的场合，有时也称为"流水线空泡。"

一般来讲，直到流水线晚期之前，指令不会有什么不可撤回的效果。所谓流水线晚期通常指从高速缓存命中时加载的数据到达前后的某个时间。在早期的简单流水线的 CPU 中，指令仅当跟在一条产生异常的指令之后才会被无效化。后来的 CPU 更广泛的采用这个技巧—例如，实现分支指令的"可能"变体。

NVRAM: 非易失性的 RAM，泛指一切断电后仍然保持内容的可写的存储器。

objdump: 一个解码目标文件并且打印信息的实用程序的典型名字。

目标代码(object code): 一种特殊的文件格式，包含有编译后的程序代码和数据，很容易转换成可执行文件格式。

目标代码库(object code library): 一个包含若干个（单独编译的）目标代码模块的文件，同时还有一个索引给出每个模块输出的公用函数和变量名。系统链接器可以接受库以及简单的目标模块，并且仅链接库中真正用到的那些模块。

八进制(octal): 以八为基数表示的数，传统上在前面加一个零。实际上，汇编器极有可能把书写时以零打头的整数解释为八进制。

乱序执行(OOO, out of order): CPU 的一种实现技术。OOO CPU 按顺序取出指令，但是指令的执行随着数据到达按照数据流顺序（只要关键队列未满）。要提供正确的顺序的语义，这种 CPU 必须保证完成的指令按顺序退出，而且任何执行中的指令的副作用都可以取消。

这听上去非常复杂，但数据流次序允许更多的指令并行执行，结果的性能增强是如此之高，以至于这种技术在非常高档的 CPU 用得极为普遍。嵌入式 CPU 还有可综合的 CPU 领域还没有实现 OOO，但看起来将来会。MIPS 体系结构清晰的寄存器到寄存器组织和缺乏条件码使得 OOO 实现起来略微容易一些。

操作码(op-code): 指令的二进制表示中部分域，对给定的指令助记符除了寄存器选择部分、嵌入的常数等等之外该域保持不变

操作数(operand): 一个操作用到的值。

优化器(optimizer): 编译器的一部分，将程序的一种正确的表示形式变换为不同的等价表示，(希望）要么节省空间要么运行得快些。

OS: 操作系统。

溢出(overflow): 当操作的结果太大无法用输出格式表示称为溢出。

填充(padding): 由于编译器需要满足硬件对数据对齐的要求，而对数据结构在存储器中的表示之间留出的空间。

页(page): 一个存储器块，通常大小为二的某次幂并且自然对齐，这是地址转换系统使用的存储器单位。大多数 MIPS 操作系统采用 4 KB 固定大小的页，但是硬件有时能够混合几种不同大小的页转换。

页色(page color): 一个页的颜色是指虚拟页地址的低 1–4 位的值。在一个虚拟索引，物理标签的高速缓存（MIPS CPU 上常用的）中，物理地址相同但是虚拟地址具有不同颜色的数据会用不同的高速缓存位置，导致高速缓存重影*(cache alias)*。

但是如果到同一物理地址的映射颜色相同就可避免重影。在出现多重映射的最常见的情况下，许多时候让操作系统负责维护同样的页色是相当自然的。不幸的是，在某些生僻的情形，问题要难得多。

页故障(page fault): 一个操作系统概念，指程序访问了一个没有分配有效的物理页地址转换的地址这种事件；在这样的操作系统中，页故障的解决要经过这几个步骤：取出适当的内容、分配物理存储器、设置地址转换、最后在出故障的指令处重新开始程序。

页表(page table): 典型的操作系统的 TLB 未命中异常处理程序要保存大量的页地址转换（接近于 TLB 可以直接用的格式），用虚拟地址的高位作为索引。这样一个数据结构称为页表。

分页的(paged): 一个存储器管理系统（比如 MIPS 的），其中采用固定大小的页（MIPS 的大小为 4 KB）为单位进行映射；地址高位进行转换，低位不变直接传递。

PageMask 寄存器(PageMask register): MIPS 存储器管理系统使用的寄存器；见第 6 章。

参数(parameter): 当谈到例程时，有些程序员说传递参数给子程序，有些（按照 C 编程手册）说传递参量给函数。他们说的都是一回事。

奇偶校验(parity): 最简单的错误检测。给字节或者其它数据单位加上一个冗余的“校验位”使得为 1 的位的总数（包括校验位）为奇数（奇校验）或偶数（偶校验）。

部分字、部分双字(partial-word, partial-doubleword): 宽度小于一个字或者双字但是硬件可以作为一个单位进行传输的数据。在某些 MIPS CPU 上，可以在一个到七个字节之间。

Patterson, David: 从 MIPS 的角度来看，他是 Hennessy 教授的亲密伙伴和共同作者（见 Hennessy and Patterson）。在 MIPS 领域之外，David Patterson 名气可能和 Hennessy 并列，他领导了 Berkeley 的 RISC 项目，后来成为 SPARC 的前身。

程序计数器 PC(prgram counter): CPU 当前正在执行的指令的地址的简称。

PC 相对寻址(PC relative): 如果一条指令使用的地址编码为相对于该指令自身位置的偏移量，那么就称该指令是 PC 相对寻址的。模块内部 PC 相对寻址的分支很方便，因为当整个模块在存储器中平移时不需要修正；这朝着 PIC（位置无关代码）前进了一步。

PCI: 大约 1993 年发明的 PC 的 I/O 总线，现在作为连接 I/O 控制器和计算机的通用方式。

PDP-11: 1970 年代全世界最受欢迎的小型机，由 DEC 制造。其影响极为深远，因为良好的设计决策和优秀的文档令其成为程序员最愿意玩弄的东西。尤其是 Ken Thompson 玩弄之后发明了 UNIX，影响了整个历史。

PDtrace: EJTAG 调试单元的一个扩展，可以收集跟踪信息，一系列经过压缩的地址允许外部软件重新构造程序的整个执行路径。要实时的保存足够多的跟踪信息是个挑战。见第 12.3 节。

窥孔优化(peephole optimization): 一种优化方式，该方式通过识别特定模式的指令序列并用更短更简单的模式进行替换来达到优化的目的。窥孔优化对于 RISC 并不那么重要，但是对 CISC 非常重要，因为提供了编译器利用复杂指令的主要机制。

页帧号 PFN(page frame number): 物理地址的高位，是分页式 MMU 的输出。

Philips: 欧洲电子业的巨头，曾经设计和制造过 MIPS 核，现在继续使用 MIPS 核。

物理地址(physical address): 出现在 CPU 的输出引脚上的地址，被传输到主存和 I/O 系统。与程序地址（虚拟地址）不同。

物理高速缓存(physical cache): "采用物理索引和物理标签的高速缓存"的简称，就是说这两个功能都采用（转换后的）物理地址。

PIC: 见位置无关代码*(position independent code)*。

流水线(pipeline): 关键的体系结构特性，通过它可以实现几条指令并行推进；见第 1.1 节。

汇编语言对流水线的隐藏(pipeline concealment by assembly): MIPS 汇编语言通常不需要程序员考虑流水线，尽管机器语言必须考虑。汇编器将代码前后移动或者插入 nop 以防意外的行为。

流水线遇险(pipeline hazard): 见遇险*(hazard)*。

流水线停顿阻塞(pipeline stall): 见停顿*(stall)*。

流水线阶段(pipestage): 具体指 MIPS 流水线的（常见为五个）阶段之一。更一般的情形，在任何流水线设计中，一个流水线阶段指的是影响任意两个时钟边沿之间指令的逻辑。

pixie, pixprof: 早期 MIPS UNIX 系统中有过重要历史影响的剖析工具。用来测量和给出高速时程序的逐条指令的行为。pixie 的工作时把原始的程序二进制转换成一个包括统计每个基本块（一个基本块就是以分支指令或分支目标隔开的一段指令）的执行次数的测量指令的版本。

pixprof 吃进 pixie 运行时产生的海量难以消化的计数数据，经过咀嚼后变成有意义的统计结果。也许有一天，这些工具或者类似的工具在别的工具包中也会提供。

PlayStation: Sony 1995 年的游戏机，由 32 位的 MIPS 微处理器驱动。

PMC-Sierra: PMC-Sierra 是一个产品线广泛的网络设备公司。它于 2000 年收购了 MIPS CPU 专业设计公司 QED，此后生产了若干 MIPS 产品。QED 延续下来的高性能定制 CPU 产品线一直继续到双 64 位 R9000X2 CPU，后继的产品采用了 MIPS 公司的核。

移植/可移植性/可移植的(porting/portability/portable): 调整为一种计算机设计的程序让它能够运行与另一计算机上。一个易于移植的程序称为可移植的，你可以根据程序的可移植性对其评定等级。

位置无关代码(position indenpendent code): 不论至于程序地址空间什么地方都能够正确执行的代码—值得一提的是，这对 Linux/MIPS 应用软件来说是必需的。见第 16 章。

可以通过简单地把所有引用都改成 PC 相对寻址的形式得到较弱的 PIC 形式。

POSIX: IEEE 为操作系统提供的编程接口制定的一个标准。整个的标准有些笨重，但是某写子集（比如众所周知的多线程标准 IEEE 1003）已经成为事实上的标准。

PostScript: 一种计算语言，同时也是表示打印页面的一种数字方式。一个极为高明的想法，最初源于 Xerox Parc，其未能统治世界基本上是由于 Adobe System 公司认为不让它进入大众市场能够赚更多钱。

pragma: C 编译器的 `#pragma` 预处理语句，用来从源代码内部选择某些编译器选项。虽然得到了 ANSI C 标准的支持，但是不够美观和灵活：GCC 倾向于将有用的选项吸收为语言的扩展。

精确异常(precise exception): 异常发生之后，指令序列位于 EPC 指向的那条指令之前的指令全部完成，而其后出现的指令看上去与完全没有出现一样。MIPS 体系结构提供精确的异常。

精度(precision): （指数据类型）数据表示所用的二进制的位数。

抢占(preemption): 抢占是因为某些更重要的事情出现而不得不停下手头正在干的事情。在计算机系统中，“更重要的事情出现”的最终原因肯定是个中断，所以抢占式的操作系统是在任何中断时都允许重新调度的。老的 Linux 版本（2.6 之前）不是抢占式的：如果中断发生时代码正好运行于核心态，在中断代码自愿停止或者自愿返回到用户态之前不允许随后重新调度。

预取(prefetch): 在不影响正在运行的程序的条件下，开始将被寻址的数据取到高速缓存的操作。如果程序员可以在数据真正加载之前以某种方式提前进行预取，那么有可能显著的提升性能。

预处理器(preprocessor): 见 C 预处理器。

PRId 寄存器(PRId register): 告诉你 CPU 型号和版本号的（只读的）CPU 控制寄存器。不应当过多依赖它。

一级 L1 高速缓存(primary(L1) cache): 在拥有多级高速缓存的系统中，指离 CPU 最近的高速缓存。

特权级(privilege level): CPU 能够运行安全的操作系统的必须能在两种不同的特权级上运行。已知的所有 MIPS 操作系统只用两个特权级：核心特权级和用户特权级（管理特权级被普遍忽略）。

用户特权级程序不允许相互干扰或干扰拥有特权的内核程序；拥有特权的程序必须正确工作。

违反特权级(privilege violation): 一个程序试图执行不允许的操作，这会导致一个异常。操作系统必须决定作出怎样的惩罚。

探头、调试(probe, debug): 见 *debug probe*。

进程(process): 一个 UNIX 的词，指对应于命令行上一个词或者单个应用程序的一块计算；包括一个控制线程、一个运行的程序的二进制代码以及可以在其中安全运行的自己的地址空间。

剖析(profiling): 带上一些收集资源使用和运行信息的工具来运行程序。那个工具可以使纯软件，在软件帮助下通过周期性中断获取信息，也可以是纯硬件（如上面的 PDtrace）。

程序地址(program address): 从软件工程师的角度看到的地址，由程序生成。也叫做虚拟地址。

首部(prologue): 见*函数首部(function prologue)*。

可编程只读存储器 PROM(programmable read-only memory): 不严格的指任意只读的程序存储器。最初是指那些在生产后可以编程的存储器，但是现在硬编程的（“掩膜”）ROM 在采用 MIPS CPU 那样大小的系统中很少见了。

有保护的操作系统(protected OS): 一个在低特权级执行任务的操作系统，这样可以防止任务进行破坏性的工作。

PTE: 页表项。在 MIPS 上就是由软件维护的用来构造硬件 TLB 数据项的数据。

PTEBase: MIPS Context 或 XContext 寄存器的一部分，装入的典型数据是一个指向存储器中地址转换的页表的指针。

Pthreads: 见 POSIX。

QED: 量子器件设备有限公司，1990 年代以来最多产的 MIPS CPU 设计组。

四倍精度 128 位浮点数(quad-precision (128-bit) floating point): MIPS 硬件不支持，但是在 n64 ABI 中作了定义，有些编译器用软件实现。在 C 语言中，这称为 `long double`。

R2000, R3000: 第一个商业化的 MIPS CPU，封装起来采用外部静态 RAM 作为一级高速缓存。

ra 寄存器(ra register): CPU 寄存器 $31，传统上用作子程序返回地址。ISA 支持这种用法，jal 指令直接用它（其 26 位目标地址域没有留下足够的空间指定哪个寄存器接受返回地址值）。

随机存取存储器 RAM(random access memory): 既可读又可写的计算机存储器。见 ROM。

Random 寄存器(Random register): 带有 TLB 时的 CPU 控制寄存器。该寄存器自动连续递增，作为 TLB 的伪随机替换算法。

Raza 微电子有限公司(Raza Microelectronics, Inc): 相对较新的进入 MIPS 的公司，Raza 制造了 XLR 族多线程、多核 MIPS64 CPU 应用于近距离网络，可能是目前所有的 MIPS 设备中吞吐量最高的。绝对值得注视!

读优先(read priority): 由于写缓冲器的原因，CPU 可能想同时执行一个读操作和一个（延迟的）写操作。先执行读操作不仅可行还能提高性能。如果 CPU 总是等待被读取的数据，那么这种条件称为读优先。但是当被读取的位置受到等待的写入影响的时候，这样做会造成一致性问题，所有很少的 MIPS CPU 尝试这么做（LSI 的 LR33000 是个例外）。

寄存器重命名(register renaming): 一种实现高性能计算机的技术，允许指令打乱正常次序执行但是执行次序对程序员保持不可见。用于 MIPS R10000。

可重定位的目标模块(relocatable object module): 一段包含必要信息和记录的目标代码，让程序能够找到并随后修改所有的偏移量和隐藏地址，把模块定位到内存的特定位置。

重定位(relocation): 为了使二进制目标代码能在不同的存储器位置执行而进行的转换过程。

再次规格化(renormalization): 在浮点计算之后，结果可能不再是规格化的。再次规格化就是重新让结果规格化。

复位(reset): 本书中用来指当你激活 CPU 的 Reset 输入时发生的事件；这在刚上电或者系统重新初始化时发生。

精简指令集计算机 RISC(reduced instruction set computer): 本书中泛指为了实现简单的流水线而设计的一类计算机体系结构。在 1980 年代后半期开始出现。

读改写 RMW(read-modify-write): 对任意类型的存储位置常见的操作序列。当 RMW 序列在读和写之间被打断时就会发生多线程系统常见

的错误，此时被中断线程可能用一个过时的值重写中断线程的值。就是说，RMW 常常有*原子性* 的要求。

只读存储器 ROM(read-only memory): 不能写入的存储设备。(在现代，更多的指不能以通常的操作写入—常常有一些离线的或者非常规的手段可以进行重新编程。

舍入模式(rounding mode): 定义浮点操作的精确行为。可以通过浮点状态/控制寄存器配置（见第 7 章）。

实时操作系统 RTOS(real-time operating system): 一个被滥用的词。几乎每个比 Linux 简单的操作系统都被称为 RTOS。真正为了帮助系统满足确定的时限而设计的简单操作系统如今可能被叫做“硬实时”操作系统。

s0–s9 寄存器(s0–s9 registers): 按照约定作用域为函数范围的变量所用的 CPU 通用寄存器的集合（$16–$23 和 $30）。凡是修改这些寄存器的函数都必须先保存它们的值。

沙盒(sandbox): 具有特殊保护的一组资源（磁盘、文件空间、存储器及 CPU 时间），在其中可以安全运行不信任的程序。这是因特网上最佳技术名词之一。

SandCraft Inc.: 一家 MIPS CPU 设计机构，成立于 1996 年，致力于全定制的 CPU 设计。其部分人员由 SGI 的 R4300 项目组的核心成员组成。SandCraft 设计了 NEC 的 Vr54xx 系列 CPU。SandCraft 于 2003 年被 Raza 微电子公司收购。

饱和算术(saturating arithmetic): 当运算结果的精度超出了表达结果的格式所能表示的范围之外时，生成可表示范围内最接近运算结果的值来处理溢出的算术运算操作成为饱和算术。对于有符号数运算，截取的结果是可表示的最大或者最小的数，对于无符号数运算，结果为负值时用零取代。

对于可能溢出的计算，通过软件代码进行检测不现实的情况，这个选择要比“自然”的回绕好得多。

条件存储(sc, store conditional): 见*连锁加载(load-linked)*。

标量(scalar): （不同于数组或数据结构的）简单变量。类似的，一次对单块数据进行操作的 CPU 称为标量机。这个词本来是为了和向量处理器的 CPU 区别的，后者可以一次操作一整块数据。

调度器(scheduler): 在多任务系统中，调度器就是决定下次执行哪个任务的程序。

SDE-MIPS: Algorithmics 针对 MIPS 目标机的开发工具包，基于 GNU C 构建的。

SDRAM, DDR DRAM(synchronous DRAM): 带有各种各样的基于沿着数据线串行数据传输的高带宽数据接口的内存芯片。

二级高速缓存(secondary L2 cache): 在带有多次高速缓存的系统中，是离 CPU 第二近的高速缓存。

区(section): 用来从程序中分开代码、各种数据、调试信息等等并且传送到目标代码中去的分块的名字。最后以每个区为单位决定其在内存中的位置。

段(segment): 见 *kseg0, kseg1*。

分段(segmentation): 一种过时的存储器地址转换和保护的方法，其中程序地址加上一个基地址来修改。x86 用过这种方法，但是自从 386 以后就没必要了。

信号量(semaphore): 一个强有力的概念，用于组织设计稳健的多任务或多处理器系统间的合作。

.set 汇编语言控制命令(.set，assembly language controls): 见 noat 等。

组，高速缓存(set, cache): 见*cache set*。

组相联(set-associative): 见*cache, set-associative*。

SGI(Silicon Graphics, Inc): 1990 年代基于 MIPS 的计算机供应商的老大，也是 MIPS 体系结构的监护人。本书写作时该公司破产了。

short: C 语言的一个整型数据类型的名字，至少跟 char 一样大但最大不超过 int。在 32 位和 64 位体系结构上，short 似乎总是代表 16 位整数。

SiByte: 一家 CPU 设计公司，在 1990 年代后期创建了 64 位 MIPS CPU。Broadcom 公司收购它之后，仍然使用 SB-12xx CPU。

信号(signal): UNIX 之类的系统中发送给普通程序的原始中断。在 Berkeley UNIX 中得到改进，由 POSIX 工作组整理并规范化，用一种简洁的方式表示多任务系统中的简单事件通信。

半导体厂商(silicon vendor): 在 MIPS 世界，指制造和销售 MIPS CPU 的任一家公司。

单指令多数据流 SIMD(single instruction multiple data): SIMD 指令对多个数据重复同一操作。现代的 SIMD 指令典型的做法是从一个宽寄存器的“切片”取得多个操作数。

SmartMIPS ASE: 一个指令级扩展，目标市场定位于“智能卡”—最大任务是进行加密/解密的极低功耗的 CPU。增加了少量的计算指令和一些存储器管理的改动对那些面向安的全应用提供更细粒度的保护。

对称多处理 SMP(symmetric multiprocessing): 大规模多 CPU 系统最为成功的一种形式，其中若干共享存储器（典型的做法是采用一致性的高速缓存）的 CPU 运行同样的操作系统内核。Linux/MIPS 已经可以直接运行于适当的 SMP 系统硬件上了。

侦听(snooping, snoopy): 见 *cache, snooping*。

片上系统 SoC(system on a chip): 用多个子系统构建的复杂器件。越来越多的是来自不同授权的多个子系统构成。

软复位(soft reset): 在数字电子中，复位是普遍可见得让一切都回到起点的信号。对于一个 CPU，复位代表了瞬间转世轮回—在几个毫秒之内死而复生。有时候，你想在复位 CPU 的时候让它记住前世的一些东西—这就是“软复位。” 见第 5.9 节。

软件指令模拟器(software instruction emulators): 一个模拟 CPU/存储器系统操作的程序。可以用来检查特别底层的软件，因为太底层的无法兼容调试器。

软件中断(software interrupts): 通过设置 Cause 寄存器的位而引发的中断；软件中断仅当这些位未被屏蔽时才会发生。

Sony: 消费类电子产品公司，其 PlayStation 采用了 MIPS 芯片。

源代码级调试器(source-level debugger): 以源程序（指令行、变量名、数据结构）形式揭示当前程序状态的调试器。源代码级调试器需要访问源代码，这样当工作于嵌入式系统软件时，调试器必须运行在宿主机上，从运行于目标机上简单的调试监控程序获得程序状态信息。

sp 寄存器/栈寄存器(sp register/stack pointer): CPU 寄存器 $29，传统上用作栈指针。

SPARC: Sun 公司的 RISC CPU 体系结构。直接源于加州大学伯克利分校的 RISC 项目，而 MIPS 出自斯坦福大学。斯坦福（位于旧金山半岛）是一所私立大学有些保守；伯克利（位于海湾对面）是所公立大学，比较激进。在微处理器设计上有许多社会历史。

稀疏地址空间(sparse address space): 一些操作系统策略（最显著的就是采用一个对象的地址作为长期的句柄）只有在你拥有比实际需要大得多的地址空间时才能工作。这样你可以承受将东西稀疏的散布于各处并且毫

无顾忌的分配空间，如稀疏地址空间的要求。现在还没有哪个稀疏地址空间的操作系统获得商业上的成功。

猜测执行(speculative execution): 一种 CPU 实现技巧，其中 CPU 在真正知道该执行哪条指令（最常见的是，在还未作出是否会发生条件分支的判断）之前就执行指令。从 R10000 以后的高端 MIPS CPU 以及越来越多的高端嵌入式 CPU 都采用该技术。

回旋锁(spinlock): SMP 系统用的一种底层的信号量形式。其特征是一个在该信号量上阻塞的线程将进入一个持续检查加锁变量的循环。这在获得一个锁只用少数几个周期时有意义。

SR 状态寄存器(SR register): CPU 状态寄存器，特权的控制寄存器之一。包含 CPU 模式的控制位。详情见第 3.3 节。

栈(stack): 后进先出的数据结构，用来记录正在运行最有意思的语言的 CPU 的执行状态。

栈参数结构(stack argument structure): 一个概念上的数据结构，第 11.2.1 节用来解释按照 MIPS 约定参数是怎样传递给函数的。

栈回溯(stack backtrace): 调试器的一个功能，对程序栈的状态进行解析以给出到达当前执行位置的函数调用的嵌套结构。该功能完全取决于严格遵循栈的使用约定，这点汇编程序必须用标准伪指令标出。

栈帧(stack frame): 一个具体函数所用的栈空间片断。

栈下溢(stack underrun): 当试图从一个从未压入数据的栈中弹出数据时出现的错误。

过时数据(stale data): 一个名词，用来指已经被最近的写操作取代但是仍然还在的数据。可以是当 CPU 已经更新了高速缓存的拷贝还未回写时的存储器中的数据；也可能是存储器内容已经被 DMA 设备替换但还未作废高速缓存时位于高速缓存中的数据拷贝。使用过时数据是个错误。

停顿(stall): 当 CPU 为完成工作而等待某些资源时流水线冻结（所有指令状态都不能推进）的状态。

独立软件(standalone software): 运行时不需要任何操作系统或标准运行环境的软件。

斯坦福(Stanford): 旧金山地区的一所大学，Hennessy 教授在那里领导了 MIPS 学术项目，也是 MIPS 公司诞生的地方。

静态变量(static variable): C 语言名词，指在编译时分配了固定的存储器地址的数据。

状态寄存器(Status register): SR 寄存器的另一个名字，见上。

stdarg: ANSI C 批准的标准宏包，提供了对带有不定个数参数或者类型只有在运行时才能确定的参数的函数的支持，但是隐藏了实现细节。

strcmp: C 语言库函数，比较两个（以空字符结尾的）字符串。

strcpy: C 语言库函数，拷贝一个（以空字符结尾的）字符串。

超级计算机(supercomputer): 口语中常用来指为了追求数值计算或计算密集型应用的性能而不必考虑成本建造的计算机。通常是通过采用一些象向量浮点指令等相对少见的体系结构特性来达到高性能的。

超流水线 CPU(superpipelined CPU): 既然流水线是个好事，那么也许通过挤压时钟周期和把单个执行阶段分为更小的片断，其中每个片断都能挤进压缩后的时钟周期—这就是超流水线。

MIPS R4000 CPU 有一点超流水线，经每个取指和数据高速缓存访问分成两个阶段并且去掉半时钟周期已得到一个八级流水线。然而加长的流水线导致分支代价增加（分支预测可以缓解这点）以及加载-使用延迟增加。

R4000 确立了这一点，就是在很广泛的 RISC 类的体系结构中，不应当使用超过五级的流水线，除非加入了分支预测特性并以某种方式解决了数据加载到使用之间的延迟问题。

超标量(superscalar): 一种 CPU 实现技术，可以同时发射多条指令。理想是复制足够多的流水线阶段来允许充分多的指令一起执行，让你“花一倍的代价得到两倍的性能”。

结果证明，在支持 OOO 组织之前，超标量 CPU 对编译出的代码只能提供不多的性能提升。对于手工编程仔细调整过的代码序列效果可能会更好。

但是有些类别的指令（例如浮点）的并行发射和运行的成本相当低，此时这种技巧能节省成本。然而，它的流行程度比其价值更高。

管理特权级(supervisor-privilege level): 介于核心和用户之间的特权级。在 MIPS32/64 中是可选的，许多 MIPS CPU 也都有实现，但是从来没被用过。见第 3.3.1 节。

交换器(swapper): 见 *byte-swapper*。

sync，存储器同步防护(sync, memory synchronization barrier): 一条让程序员明确表示程序中的读写次序真的很重要的指令。程序顺序中在 sync 指令之前的任何读写必须完成之后，才能执行 sync 之后的读写。

同步逻辑(synchronous logic): 按照在相继的寄存器之间插入的“组合逻辑“阶段，每个寄存器在固定的全局时钟信号转换期间存储信息。将 Verilog 代码变成能用的芯片逻辑的编译器，大多数目的，被局限于映射到同步逻辑的 Verilog 代码，那是所有已知的可综合的 CPU 的工作方式。

对能用得异步逻辑进行可靠的综合是一个极端困难备受挫折的目标，如果成功可以产生性能功耗比的极大提升。

合成指令(synthesized instructions): 见 *instruction synthesis by assembly*。

系统调用(syscall): 一条为了产生异常而存在的指令。异常充当一个对内核的安全的子程序调用。syscall 指令有一个空闲的硬件不作解释的域，软件可用它编码不同的系统调用类型。但是 Linux 并不用它—更愿意用一个在通用寄存器中传递的值来区分系统调用类型。

t0–t9 寄存器/临时寄存器(t0–t9 register/temporaries): CPU 寄存器 $8–$15 和 $24–$25，按约定用于临时变量；任何函数都可以用这些寄存器。缺点是这些值不能保证在系统调用之间保存。

TagHi、TagLo 寄存器(TagHi, TagLo registers): MIPS32/64 的 CP0 寄存器；它们是高速缓存标签内容的补给站。

TC: 在 MIPS MT(多线程) CPU 中，TC 是运行线程的硬件单元，拥有一个 PC 和一组通用寄存器。

临时寄存器(temporary registers): 见 *t0*。

L3 三级高速缓存(tertiary (L3) cache): 位于 L2 和内存之间的第三级高速缓存。

抖动(thrashing): 一种启发式优化的崩溃，其特征是不断的重复失败。“高速缓存抖动”是一种特例，其中程序频繁使用的两个地址为竞争同一个高速缓存存储块，反复将对方替换出高速缓存，使得高速缓存失去作用。在软件方面花费了很大努力试图构造出不易受高速缓存抖动代码，都以失败而告终；但是使用组相联高速缓存后这个问题或多或少的消失了。

线程(thread): 线程就是程序中按照程序员意图顺序执行的部分。这是软件多任务操作系统、多处理器系统或多线程 CPU 硬件的基本构造单元。

定时器(timer): 为 CPU 提供的一个功能，是一个固定速度的计数器，该计数器带有一种机制，能在计数到达某个指定值时会引发中断。

转换旁视缓冲器 TLB(translation lookasdie buffer): 将程序页地址转换为物理页地址的相联存储器。当 TLB 不包含你需要的地址转换时，CPU 发生异常，由系统软件加载合适的地址转换数据，然后返回并重新执行出错的地址引用。见第 6 章。

TLB, 禁锢项(TLB, wired entries): 前几个 TLB 数据项可以设置成不受 TLB 重填处理程序的随机替换策略的影响：把 wired 寄存器设置到零以上的一个值，random 寄存器就不会取等于或小于 wired 的值。很老的 MIPS CPU 没有 wired 寄存器总是保留前八项。

TLB 无效异常(TLB invalid exception): 当 TLB 项匹配相应地址但是自身被标记为无效时发生的异常。

TLB 落空(TLB miss): 当没有 TLB 项匹配程序地址时发生的异常。大多数 TLB 未命中异常用一个独特专用的异常入口点。这样做是因为这个异常是操作系统中出现得最多的，省去识别异常的代码可以节省时间。

很老的 MIPS CPU 只对用户态的 TLB 未命中异常使用专门的异常入口点：MIPS32/64 CPU 对异常模式（一个少见但是特殊的情形）之外的 TLB 未命中都用专门的异常入口。

TLB 修改异常(TLB modified exception): 一个 TLB 项匹配了一个写操作的地址，但该 TLB 项标记为不可写时发生的异常。

TLB 检测，tlbp (TLB probe, tlbp): 一个指令，可以提交一个程序地址给 TLB 来查看是否存在某一项能够转换该地址。

TLB 重填(TLB refill): 在 TLB 未命中之后向 TLB 添加新的数据项的过程。

工具链、工具箱(toolchain, toolkit): 用来从源代码开始生成可运行的程序所需要的全部工具的集合（编译器、汇编器、链接器、库等等）。

东芝(Toshiba): 一家日本的芯片制造商，拥有 MIPS 授权。东芝作为普通销售的 CPU 器件供应商并不突出，成名的是作为 Sony PlayStation 2 的“感情引擎”的心脏：一个 64 位的浮点向量处理器，不比 1980 年代的任何超级计算机逊色。

转换地址或地址区(translated address or address region): 必须经过 TLB 转换（否则出错）的 MIPS 程序（虚拟）地址。包括 kuseg 区，用户特权级软件必须在此运行；还有映射的核心特权级区域 kseg2。64 位 CPU 拥有更多的转换区。

转换旁视缓冲器(translation lookaside buffer): 见 *TLB*。

自陷(trap): 影响特定指令的内部事件导致的异常。

取整(trunc): 浮点指令 trunc 把一个浮点数舍去小数部分得到一个整数。

TTL: 晶体管-晶体管逻辑（transistor-transistor logic）的首字母缩写。这是一个信号约定，用来确定一个电气信号代表 1、0 还是介于二者之间未定义。TTL 基于早期的 5 V 逻辑族的习惯。TTL 信令至少到 1990 年代后期在所有的微处理器系统上通用；其最可能的替代者是略微修改后采用 3.3 V 供电电源。

两路组相联(two-way set-associative): 见*cache, set-associative*。

通用异步收发器 UART(universal asynchronous receiver/transmitter): 一种串口控制器。

Ultrix: DEC 为其运行在基于 MIPS 的 DECstation 计算机上的 BSD 系列操作系统起的商标名。Ultrix 运行在 DEC 要求的小尾端上，使得它与当代的 MIPS 工作站和服务器不能二进制兼容。

UMIPS: 见 *MIPS UMIPS 4.3BSD*。

未对齐地址访问异常(unaligned access exception): 由于引用（load/store 字或半字）未能适当对齐的地址而导致的自陷。

未对齐的数据(unaligned data): 存储在内存中的数据，但不保证位于对齐到适当的地址边界。未对齐的数据只能用特殊的指令序列才能可靠访问，见 8.5.1 节。

不可高速缓存的(uncachable): CPU 读写不经过也不影响高速缓存的存储区。对于 kseg1 区域和 TLB 项标记为不作高速缓存的进行地址转换的区域都是如此。

不用高速缓存的(uncached): 不搜索或不写入高速缓存的 CPU 读写。

下溢(underflow): 当浮点操作的结果值太小无法正常表示时成为下溢。

统一高速缓存(unified cache): 对于所有 CPU 访问都搜索和更新的高速缓存，包括取指和引用数据。基本上所有 MIPS CPU 都采用单独的一级 I- 和 D- 高速缓存，但是大多数 L2 和 L3 高速缓存都是统一的。

未实现的指令异常(unimplemented instruction exception): 当 CPU 不能识别指令码时发生的异常；也用在不能完成浮点指令想要软件方针的地方。

联合体(union): C 语言的数据声明，表示该数据对不同的类型有不同的解释。如果你存储数据时用一种类型，读出时用另一种类型，其结果以有趣的方式显示出高度的不可移植性。

单处理器(uniprocessor): 不跟别的处理器共享存储器的 CPU。

(UNIX-like): 类似真正的 UNIX 的系统，但是没有版权或者正式的标准。包括 Linux，还有来自各个 OpenBSD 和 FreeBSD 组的操作系统，以及象 Sun 的 Solaris 或者 SGI 的 Irix 之类的商业操作系统。

未映射、不作转换的(unmapped, untranslated): 指 kseg0、kseg1 地址区。

展开的循环(unrolled loop): 程序中的循环，经过变换后（大多数时间）在跳转之间执行超过一次循环的工作。常常能使程序运行地快一点；有时由高级的编译器自动进行这种变换。

用户空间(user space): 用户特权级可访问的地址空间（kuseg）。

用户区(userland): Linux 用来称呼系统处于内核之外的部分：库还有基本应用程序。GNU/Linux 操作系统的用户区大多来自于 GNU 项目。

用户特权级(user-privileged level): MIPS CPU 的最低特权级状态，其中只能用常规指令集，程序地址必须位于 kuseg 内。操作系统可以防止用户特权级的程序相互干扰和或者干扰操作系统。

utlbmiss 异常(utlbmiss exception): 用户态的 TLB 未命中异常。在老的 MIPS CPU 上，专用的 TLB 未命中异常入口点仅用于用户态异常。

v0–v1 寄存器(v0–v1 registers): CPU 寄存器 $2–$3，按照约定用来容纳函数的返回值。

(varargs): stdarg 现在已经过时的老版本。

VAX: DEC 具有开创性的 32 位小型机体系结构，但绝对不是 RISC。第一个支持虚拟存储器（V 源于 virtual）的小型机。

向量、向量处理器(vector, vector processor): 拥有能够一次在一整块数据集合上执行操作，特别是浮点操作指令的处理器。这是一种被称为单指令多数据（SIMD）的并行处理的例子；是第一种实际使用的并行处理。数值运算密集型的超级计算机依赖向量处理来提高速度。

可向量化的(vectorizable): 容易实现自动循环优化来利用向量或其它 SIMD 操作的程序源称为可向量化的。实际中，通常需要仔细设计的程序才能可以进行那种优化。

Verilog: 一个编程语言，用来描述对真实硬件的模拟或者“综合”的逻辑设计。大多数实际用得现代的可复用逻辑设计都是用 Verilog 写的。

虚拟地址(virtual address): 见程序地址*(program address)*。

虚拟存储器(virtual memory): 一种运行应用程序的方式，不需要实际上真的给程序认为自己需要的全部存储器，但是程序自己并不知道区别。经过安排让试图访问并不真正在存储器的部分时，导致操作系统调用。操作系统找出需要的存储器（代码或数据），修改地址映射让应用程序可以找到，然后从导致无效访问的指令处重新执行应用程序。大型的操作系统（UNIX 之类或者现代的 Windows）总是使用虚拟存储器。

VMS: DEC 为 VAX 小型机开发的操作系统。

void: 一个让 C 程序显得更清晰的数据类型，表示没有可用的值。

易变的volatile: C 或汇编语言声明的数据的属性。一个易变的变量指那种其行为不像存储器的变量（即并不是简单的返回上次存进去的值）。如果没有这个属性，优化器可能会认为没有必要重新读取一个值；若该变量代表的是你在检测的存储器映射的 I/O 地址，这样做就错了。

VPE: 在 MIPS MT （多线程）中，一个 VPE 包含一个或多个 TC（在其中运行程序）并且提供一组完整的 CP0 寄存器和其它资源，使得看上去就像是一个完整的 MIPS32/64 兼容的 CPU。

虚拟页号 VPN (virtual page number): 程序（虚拟）地址进行转换的部分。程序地址的低位（即页内地址，通常页为 4 KB）不用改变直接传递给物理地址。

VxWorks: 一个在嵌入式应用中的实时操作系统内核，由风河公司(Wind River System, Inc)开发和销售。

WatchHi、WatchLo 寄存器(WatchHi, WatchLo registers): 实现数据观察点的协处理器 0 寄存器，R4000 风格的 CPU 上都有。

观察点(watchpoint): 调试器提供的一个特性，当访问给定地址时可以暂停正在运行的程序并将控制传递回用户。NEC 的 Vr4300 CPU 拥有其中之一。

wbflush: 一个例程/宏的标准名，该例程/宏确保位于缓冲队列的所有外部写周期离开 CPU 。

Whitechapel: 曾经短暂兴起的一家位于英国的 UNIX 工作站公司，1987 年卖出了第一台 MIPS 桌面计算机。

工作站(workstation): 这里指运行 UNIX 之类操作系统的桌面计算机。

回绕(wraparound): 有些存储器系统（包括被隔离的 MIPS 的高速缓存）有个性质，访问超出存储器大小的地址时就简单回到起始地址开始访问。

写缓冲器(write buffer): 一个保存 CPU 写周期的地址和数据的 FIFO 存储单元（通常可达四个）。在存储器系统处理写操作的过程中，CPU 可以继续执行。当采用透写高速缓存时，写缓冲器特别有效。

回写高速缓存(write-back cache): 见 *cache, write-back*。

透写高速缓存(write-through cache): 见 *cache, write-through*。

XContext 寄存器(XContext register): 与 TLB（存储器管理硬件）有关的协处理器 0 寄存器。在为 64 位寻址的虚拟存储区域采用特殊组织的页表的系统上，提供一种处理 TLB 未命中的快速方法。

零寄存器(zero register): CPU 寄存器 $0，其用法极为特殊：不管写入什么值，总是返回零值。

参考资料

参考书和参考文章

Cohen,D. "On Holy Wars and a Plea for Peace." *USC/ISI IEN* 137, April 1, 1980. 网上从 Wikipedia (`www.wikipedia.org`)中检索"尾端(endianness)"就可找到该论文。

Heinrich, J *MIPS R4000 User's Manual.* Englewood Cliffs, NJ: Prentice Hall, 1993

MIPS III 体系结构的经典；内容非常详细，详细得有时很难查找到所要的信息。同时该书从一个相当严格的角度，仅仅讲述那些在 MIPS 体系结构中真正通用的领域，而省略了具体实现相关的部分。在本书写作时，该书以 PDF 格式提供下载。（参见下面列出的网上资源）

Hennessy, J. and D. Patterson. *Computer Architecture: A Quantitative Approach.* 3rd edition. San Franciso: Morgan Kaufmann, 2002.

在 MIPS 的特定领域之外，这是唯一的一本值得拥有的关于现代计算机体系结构的书。它唯一的不足就是太厚了，但是厚得有价值。对于基本概念的讲述老版本仍然不错，可以比较合理的价格拿到二手书；但是如果你觉得第三版的价格还算合理的话，你会发现该版中的新内容确实物有所值。在你购买之前，检查一下是否出了后续版本。

Kernighan,B. and D. Richie *The C Programming Language.* 2nd edition. Englewood Cliffs, NJ: Prentice Hall, 1988. 如果你想对 C 语言了解更多，那么不能没有本书。现在你也许应该买更新的 ANSI 版本了，遗憾的是书有点厚。

Love, R. *Linux Kernel Development.* Carmel, IN: Sams Publishing, 2003.

有关 Linux 内核的高级指南一部篇幅适中、写的很好的书，不论是否有源代码都可以阅读本书。本书善于解释一件事情为什么那样做而不是这样做。

Rosenberg, J. *How Debuggers Work: Algorithms, Data Structures, and Architecture.* New York; John Wesley & Sons, 1996.

除非你在开发调试工具，否则你可能不需要理解到该书中将的那么详细的程度，不管怎样，该书能够帮助你更加有效的使用你所拥有的调试工具。

Sweazey,P. and A.J.Smith. "A class of Compatible Cache- Consistency Protocols and Their Support by the IEEE Future Bus." *Proceedings of the 13th International Symposium on Computer Architecture*, 1986

Tanenbaum, A. and A.S.Woodhull. *Operating Systems Design and Implementation.* 3rd edition. Englewood Cliffs, NJ: Prentice Hall, 2006.

该书作为操作系统的入门书很不错。书中描述的 Minix 操作系统具有重要的历史意义，（其早期的版本）为 Linux 的诞生提供了最初的原动力。

网上资源

下面给出的 URL 在本书出版时经检查确认有效，但是与任何有关 Web 的印刷信息一样，内容有可能变化。如果你无法访问某个网址，用你最喜欢的搜索引擎查找一下该网址是否已经搬家了，还应当检查一下自从本书出版以来新的关于 MIPS 和 Linux 的信息源。

MIPS 体系结构已经出来很长时间了，从网上也许能够找到其它一些有用的读物。

Advanced Micro Device (AMD): `www.amd.com` 网址有关于从 Au1000 到 Au1550 的 MIPS 处理器的信息，在被 AMD 收购之前最初是 Alchemy Semiconductor 设计了这些处理器。从 Google 上搜索 `alchemy site:amd.com` 找出合适的网页。

Broadcom Corporation: `www.broadcom.com` 网址有关于 Sibyte 在被 Broadcom 收购之前设计的 BCM 系列 MIPS 处理器的信息。从 Google 上搜索"`communications processors`" `site:broadcom.com` 可以找到相应的页面。

GNU C Compiler 的网址在 `gcc.gnu.org`。在 `gcc.gnu.org/onlinedocs` 拥有当前版本和老版本的在线手册。

Linux MIPS port: 负责该工作的小组维护的网站在 `www.linux-mips.org`。该站点总是提供最新的源代码 Linux 内核在 MIPS 上的移植。此外，还有一些其它有关移植和构建内核的支持辅助信息。该站点还可以帮助你找出有关处理器、系统、工具链等等的更多信息，有些部分提供了历史和背景信息。

MIPS Technologies: MIPS 体系结构的监护人，通过向其它厂商发布 CPU 核的授权来谋生，其网址在 `www.mips.com`。你可以在那里发现到许多与 MIPS 有关的公司的连接。

PMC-Sierra Inc.: 位于 `www.pmc-sierra.com` 的网站上有关于 PMC-SIERRA 的 MIPS 设备的信息，包括许多本来由 QED 公司设计的处理器，后者被 PMC-Sierra 兼并。

The Linux Devices Web site: 该站点位于 `www.linuxdevices.com`，广泛提供有关在嵌入式系统中使用 Linux 的新闻和信息。

The Linux Kernel Archives: 这些内容位于 `www.kernel.org`。从这里总可以下载到最新版本的 Linux 内核源代码（凡是存在有移植的各种体系结构，该站点都提供服务）。最近的版本通常也保存在这里，如果最新的版本还没有移植到你的硬件上时，这些版本就用得上了。

译者补遗

这里针对本书中未提到或未详细解释，但在阅读编译器生成的汇编输出时可能遇到的与 PIC 有关的汇编指示，做一简单解释。这些指示名字一般以 **.cp** 开头，cp 代表 context pointer(上下文指针，或译为境遇指针)，通俗的可以理解为就是我们熟悉的 **gp**。

.cpload 和 .cprestore

该指示后接一个寄存器（通常为 **t9** 即 **$25**）参数，一般位于函数首部，且位于 **.set noreorder** 的作用域内。它指示汇编器，发送几条指令根据参数寄存器的值加载 GOT 表的基址到 **gp**。有关 GOT 的内容，请参阅 16.2 节。

.cpload reg 指示一般要放在函数首部，展开如下：

```
lui     gp, _gp_disp
addui   gp, gp, _gp_disp
addu    gp, gp, reg
```

其中 `_gp_disp` 代表从本函数的入口点到 GOT 表入口点之间的偏移量，是一个在编译时就能确定的值。*reg* 按照约定一般为 **t9**，在本函数被调用前，由调用函数填入本函数的地址，然后通过一条 **jalr reg**指令调用。这样在本函数的入口点，*reg* 的值就是本函数入口地址，加上到 GOT 入口的偏移量，存入 **gp**，便将 GOT 表入口地址送入了 **gp**。

.cprestore offset 指示汇编器将当前的 **gp** 保存于栈上距离 **sp** 偏移为 *offset* 的位置，即生成：

```
sw  gp, offset(sp)
```

同时在每个随后的跳转调用指令（即 **jal**，注意不是 **bal**，后者属于分支调用）返回后恢复 **gp** 的值，即生成：

```
lw  gp, offset(sp)
```

指令。

.gpword 和 .cpadd

这两条汇编指示一起可以生成一个位置无关的跳转表。由 **.gpword** 创建的的地址表入口可以用 **.cpp** 加上 **gp** 得到正确的运行地址。

```
.gpword local_symbol
```

类似于 **.word**，但是链接后的值为 `local_symbol` 到 cp 的偏移量。

```
.cpadd reg
```

将 **gp** 的值加到寄存器 *reg* 上。

.cpsetup 和 .cpreturn

这两个用于 64 位，分别相应于 32 位 PIC 的 **.cpload** 和 **.cprestore**。

```
.cpsetup reg, reg2/offset, label
```

展开为：

```
sd      gp, offset(sp)
lui     gp, %hi(%gp_rel(label))
daddiu  gp, gp, %lo(%gp_rel(label))
daddu   gp, gp, reg
```

按照约定，*reg*=**t9**, `label` 为函数的入口标号。

```
.cpreturn
```

展开后成为：

```
ld      gp, offset(sp)
```

其中的 *offset* 就是上一个 **.cpsetup** 中的 *offset*。

对本书的评价

大多数体系结构方面的书，最后都差不多是该体系结构的汇编语言的一本详细而又繁琐的全面罗列。《See MIPS Run》是一个出色的反例，应当作为所有这一类书的作者的一个榜样。Dominic Sweetman 提供了对于体系结构参考书来说必要的细节，但对细节的表述总是洞察出整体上最关键的体系结构特性（及其来龙去脉）。结果造就了这本极为精彩并且饶有趣味的关于体系结构具体就是 MIPS 的书，同时清晰地阐明了渗透进体系结构发展的各种技术、经济、历史、甚至政治的因素。第二版中重要的补充就是关于操作系统、软件移植和 ABI 的内容，这些补充使得该书成为软件开发人员的优秀参考书。任何在 MIPS 体系结构上工作的人都会乐于收藏本书。

——Randy Allen, Catalytic 的创始人和技术总监

本书对于任何 MIPS 体系结构的参考手册都是一本极好的配套书。秉承第一版的传统，第二版仍然把重点放在通过具体例子阐明软硬件接口上。此外，第二版吸收了最近从 MIPS-I/V 体系结构向 MIPS32/64 体系结构的过渡的有关内容，包括了体系结构方面对多线程的支持。简而言之，本书是任何严肃的 MIPS 体系结构的程序员的一部必读之作。

——Jan-Willem van de Waerlt, Fellow
Philips Semiconductor 半导体公司

第二版不仅针对第一版作了详尽的彻底的更新，而且将最知名的 RISC 体系结构——MIPS——和最知名的开放源码操作系统——Linux——结合了起来。本书的第一部分从 MIPS 设计原则开始，然后介绍了 MIPS 指令集和编程资源。本书以 MIPS32/64 作为基准对体系结构所有的其它版本进行了比较。

与第一版相比的一个重大变化，就是一只企鹅坐在驾驶员的座位上，因为本书以 Linux 内核代码作为真实的底层操作系统的例子，并且展示了怎样在 MIPS 体系结构提供的基础上构建 Linux ——包括单处理器和对称多处理器版本。本书从操作系统的最底层——中断、存储器地址转换——开始，一直讲到 Linux/MIPS 应用程序代码加载进内存、与库函数链接并且运行的整个过程。

- 对于 Linux 在真实硬件上的运行做了最佳的讲解。
- 对于 MIPS 指令集提供了一个完整、全面的、更新的、并且易于使用的指导。
- 继续保持了使得本书第一版极具可读性的引人入胜的写作风格，反映了作者超过 20 年的在基于 MIPS 体系结构的系统上的设计经验。

Embedded System Programming,
Computer Architecture, Advanced Computing