第一章 RISCs 与 MIPS

MIPS 是高效精简指令集计算机(RISC)体系结构中最优雅的一种；即使连 MIPS 的竞争对手也这样认为,这可以从 MIPS 对于后来研制的新型体系结构比如 DEC 的 Alpha 和 HP 的 Precision 产生的强烈影响看出来。虽然自身的优雅设计并不能保证在充满竞争的市场上长盛不衰，但是 MIPS 微处理器却经常能在处理器的每个技术发展阶段保持速度最快的同时保持设计的简洁。

相对的简洁对于 MIPS 来说是一种商业需要，MIPS 起源于一个学术研究项目，该项目的设计小组连同几个半导体厂商合伙人希望能制造出芯片并拿到市场上去卖。结果是该结构得到了工业领域内最大范围的具有影响力的制造商们的支持。从生产专用集成电路核心(ASIC Cores)的厂家(LSI Logic,Toshiba, Philips, NEC)到生产低成本 CPU 的厂家(NEC, Toshiba, 和 IDT),从低端 64 位处理器生产厂家(IDT, NKK, NEC)到高端 64 位处理器生产厂家(NEC, Toshiba 和 IDT).

低端的 CPU 物理面积只有 1.5 平方毫米(在 SOC 系统里面肉眼很难找到).而高端的 R10000 处理器，第一次投放市场时可能是世界上最快的 CPU，它的物理面积几乎有 1 平方英寸,发热近 30 瓦特.虽然 MIPS 看起来没什么优势,但是足够的销售量使其能健康发展:1997 年面市的 44M 的 MIPS CPU,绝大多数使用于嵌入式应用领域.

MIPS CPU 是一种 RISC 结构的 CPU, 它产生于一个特殊的蓬勃发展的学术研究与开发时期.RISC(精简指令集计算机)是一个极有吸引力的缩写名词,与很多这类名次相似,可能遮掩的真实含义超过了它所揭示的.但是它的确对于那些在 1986 到 1989 年之间投放市场的新型 CPU 体系结构提供了一个有用的标识名，这些新型体系结构的非凡的性能主要归功于几年前的几个具有开创性的研究项目所产生的思想。有人曾说："任何在 1984 年以后定义的计算机体系结构都是 RISC"；虽然这是对于工业领域广泛使用这个缩写名词的嘲讽，但是这个说法也的确是真实的-1984 年以后没有任何一款计算机能够忽视 RISC 先驱者们的工作。

在斯坦福大学开展的 MIPS 项目是这些具有开创性的项目中的一个。该项目命名为 MIPS(主要是无内锁流水段微型计算机的关键短语的缩略)同时也是"每秒百万条指令数"的双关语。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研究的远远不够，流水线技术其实能够被更好的利用。尤其是当结合了 1980 年的硅材料设计水平时。

1.1 流水线

从前在英格兰北部的一个小镇里,有一个名叫艾薇的人开的鱼和油煎土豆片商店。在店里面，每位顾客需要排队才能点他（她）要的食物（比如油炸鳕鱼，油煎土豆片，豌豆糊，和一杯茶）。然后每个顾客等着盘子装满后坐下来进餐。

艾薇店里的油煎土豆片是小镇中最好的，在每个集市日中午的时候，长长的队伍都会排出商店。所以当隔壁的木器店关门的时候，艾薇就把它租了下来并加了一倍的桌椅。但是这仍然不能容纳下所有的顾客。外面排着的队伍永远那么长，忙碌的小镇居民都没有时间坐下来等他们的茶变凉。

他们没办法再另外增加服务台了；艾薇的鳕鱼和伯特的油煎土豆片是店里面的主要卖点。但是后来他们想出了一个聪明的办法。他们把柜台加长，艾薇，伯特，狄俄尼索斯和玛丽站成一排。顾客进来的时候，艾薇先给他们一个盛着鱼的盘子，然后伯特给加上油煎土豆片，狄俄尼索斯再给盛上豌豆糊，最后玛丽倒茶并收钱。顾客们不停的走动；当一个顾客拿到豌豆糊的同时，他后面的已经拿到了油煎土豆片，再后面的一个已经拿到了鱼。一些穷苦的村民不吃豌豆糊-但这没关系，这些顾客也能从狄俄尼索斯那里得个笑脸。

这样一来队伍变短了，不久以后，他们买下了对面的商店又增加了更多的餐位。
这就是流水线。将那些具有重复性的工作分割成几个串行部分，使得工作能在工人们中间移动，每个熟练工人只需要依次的将他的那部分工作做好就可以了。虽然每个顾客等待服务的总时间没变，但是却有四个顾客能同时接受服务，这样在集市日的午餐时段里能够照顾过来的顾客数增加了三倍。图 1.1 说明了艾薇的方法，是由她那很少涉猎非虚现实问题的儿子爱因斯坦绘制的。
如果将程序看成是内存中存储的一堆指令的话，一个即将运行的程序看起来和排着队等待接受服务的顾客没什么相似之处。但是如果从 CPU 的角度来看，就不一样了。CPU 从内存中提取每条指令，进行译码，确定需要的操作数，执行相应操作，并存储产生的任何结果-然后再次重复同样的工作。等待执行的程序就是一个等待一次一个的流过 CPU 的指令队列。
由于每条指令都要做不同的工作，因此在 CPU 内部已经配有各种不同的专用的大块逻辑电路，所以构造一个流水线并没有使 CPU 复杂度增加多少；只是让 CPU 工作负载更重一些而已。
对于 RISC 微处理器来说使用流水线技术不是什么新鲜事儿。真正重要的在于完全的重新设计-从指令集开始-目的是使流水线更加高效。因此，怎样才能设计一个高效的流水线实际上可能是一个错误的问题。正确的提问应该是，是什么使得流水线效率低下？
第二章 MIPS 体系结构

在计算世界中, "体系结构"一词被用来描述一个抽象的机器,而不是一个具体的机器实现. 这一点非常有用的, 用来区分在市场广告上已经被滥用的"体系结构"这个术语. 读者有可能不熟悉"抽象描述",但其概念其实很简单.

当然，如果你是一个喜欢在 滑的路上开快车的司机，前轮还是后轮驱动就很有所谓了。计算机也是如此。如果你需要高性能计算，一个计算机的具体参数与实现对你就很重要了。

一般而言，一个 CPU 的体系结构有一个指令集加上一些寄存器而组成。“指令集”与”体系结构“这两个术语是同义词。你经常会看见 ISA(指令集体系结构--ISA)的缩写。

MIPS 体系结构家族包含如下几代。每一代之间都有一些区别。

MIPS 1：32 位处理器使用的指令集。仍然被广泛使用着。
MIPS II：为 R6000 机器所定义的，包含了一些细微的改进。后来实现在 1995 年的 32 位 MIPS 实现中。
MIPS III：R4xxx 的 64 位指令集。
MIPC IV: MIPS III 的一个细微的升级。定义在 R10000 和 R5000 中。

上述的 MIPS 体系结构等级与 MIPS 公司提供的文档中定义的是一致的。这些文档提供了足够的信息，以使得同一个 UNIX 应用程序可以在不同的 MIPS 体系结构等级上运行，但是在操作系统或底层相关的代码方面的移植方面，显得不足。MIPS CPU 其他一些软件可见的方面都是于具体 CPU 实清 b 相关的。

在本书中，我将更加慷慨大方些。有时候，我会描述一个在 MIPS 体系结构手册中找不到的，但却在所有 MIPS III 体系结构实现中能发现的并且你会遇到的功能。

另外，除了 ISA 等级，大多数 MIPS CPU 在实现方法上分为两大类：早期的 MIRS R3000 和其他所有的 32 位 MIPS CPU；另外就是已 MIPS R4000 为代表的 64 位 CPU。

有不少 MIPS CPU 的实现加入了一些新指令和其他一些有趣的功能。对於软件或工具(如编译器)而言，要利用这些非标准的，依赖于具体实现的功能是不容易的。

我们可以在两种细节层次上来描述 MIPS 的体系结构。第一种描述(本章)是在汇编语言的层次上看待你的程序，比如，你在工作站上写一个应用程序。这也意味着 CPU 的所有一般的计算是可见的。

在下面章节里，我们将介绍 MIPS 的各个方面，包括建构在 CPU 之上的操作系统所掩盖的所有 CPU 的细节，CPU 控制寄存器，中断，陷入，高速缓冲操作和内存管理。至少我们会将一个 CPU 分成一些小部分来学习和介绍。

2.1 MIPS 汇编语言的特点

汇编语言是 CPU 二进制指令的可读写版本。我们在后面将有单独的一章来讲述汇编语言。从来没有接触过汇编语言的读者在阅读本书时可能会有一些迷惑 。

大多数 MIPS 汇编语言都是非常古板的，都是一些寄存器号码。但是工具链(toolchains)可以使得使用微处理机语言变得简单。工具链至少允许程序员引用一些助记符，而严格的汇编语言要求严格的数字编码。大多我们都是用比较熟悉的 C 预处理器。C 预处理器会把 C 风格的注解去掉，而得到一个可用的汇编代码。

有 C 预处理器的帮助，MIPS 汇编程序都是用助记符来表示寄存器。助记符同时也代表了每个寄存器的用法(我们将在 2.2 节介绍这一点)

对於熟悉汇编语言但不熟悉 MIPS 的读者，下面是一些例子。

```
/* this is a comment */
#so is this

entrypoint: #this'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。每一行的结束是一个指令的结束，并且忽略任何“#”之后的内容，认为是注释。在一行里可以有多条指令。指令之间要用分号“；”隔开。

一个符号(label)是一个后面跟着冒号“：”的字。符号可以是任何字符串的组合。符号被用来定义一段代码的入口和定义数据段的一个存储位置。

如上所示，许多指令都是 3 个操作数/符(operand)。目标寄存器在左侧(注意，这一点与 Inetel x86 正相反)。一般而言，寄存器结果和操作符的顺序与 C 语言或其他符号语言的方式是一致的。 例如：

subc $1, $2, $3

意味着：

$1 = $2 - $3;

这方面我们就先讲这么多。

2.2 寄存器

对於一个程序，可以有 32 个通用寄存器，分别为：$0-$31。其中，两个，也只有两个的使用不同于其他。

$0：不管你存放什么值，其返回值永远是零。以此可以用来做32位的清零用

$31：永远存放着正常函数调用指令(jal)的返回地址。请注意call-by-registe的 jalr 指令可以使用任何寄存器来存放其返回地址。当然，如不用$31，看起来程序会有点古怪。

其他方面，所有的寄存器都是一样的。可以被用在任何一个指令中(你也可以用$0 作为一个指令的目标寄存器。当然不管你存入什么数据，数据都消失了。)

MIPS 体系结构下，程序计数器不是一个寄存器，其实你最好不要去那样想。在一个具有流水线的 CPU 中，程序计数器的值在一个给定的时刻有多个可选值。这一点有点迷惑人。jal 指令的返回地址跟随其后的第二条指令。

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

上述的解释是有道理的，因为紧跟踪 jal 指令后面的指令，由於在 delay slot(延迟位置)上--请记住，关于延迟位置的规则是该指令将在转移目标(如上述的 printf)之前执行。延迟位置指令经常被用来传递函数调用的参数。

MIPS 里没有状态码。CPU 状态寄存器或内部都不包含任何用户程序计算的结果状态信息。

hi 和 lo 是与乘法运算器相关的两个寄存器大小的用来存放结果的地方。它们并不是通用寄存器，除了用在乘除法之外，也不能有做其他用途。但是，MIPS 里定义了一些指令可以往 hi 和 lo 里存入任何值。想一想我们会发现，这是非常有必要的当你想

要恢复一个被打断的程序时。

浮点运算协处理器(浮点加速器，FPA)，如果存在的话，有 32 个浮点寄存器。按汇编语言的简单约定讲，是从$f0 到$31。

实际上，对於 MIPS I 和 MIPS II 的机器，只有 16 个偶数号的寄存器可以用来做数学计算。当然，它们可以既用来做单精度(32 位)和双精度(64 位)。当你做一个双精度的运算时，寄存器$f1 存放$f0 的余数。奇数号的寄存器只用来作为寄存器与 FPA 之间的数据传送。

MIPS III CPU 有 32 个 FP 寄存器。但是为了保持软件与过去的兼容性，最好不要用奇数号的寄存器。

2.2.1 助记符与通用寄存器的用法

我们已经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。

寄存器编号 助记符 用法
0 zero 永远返回值为 0
1 at 用做汇编器的暂时变量
2-3 v0, v1 子函数调用返回结果
4-7 a0-a3 子函数调用的参数
8-15 t0-t7 暂时变量，子函数使用时不需要保存与恢复
24-25 t8-t9
16-25 s0-s7 子函数寄存器变量。子函数必须保存和恢复使用过的变量在函数返回之前，从而调用函数知道这些寄存器的值没有变化。
26,27 k0,k1 通常被中断或异常处理程序使用作为保存一些系统参数
28 gp 全局指针。一些运行系统维护这个指针来更方便的存取“static“和”extern″变量。
29 sp 堆栈指针
30 s8/fp 第 9 个寄存器变量。子函数可以用来做桢指针
31 ra 子函数的返回地□
’7d

虽然硬件没有强制性的指定寄存器使用规则，在实际使用中，这些寄存器的用法都遵循一系列约定。这些约定与硬件确实无关，但如果你想使用别人的代码，编译器和操作系统，你最好是遵循这些约定。

寄存器约定用法引人了一系列的寄存器约定名。在使用寄存器的时候，要尽量用这些约定名或助记符，而不直接引用寄存器编号。

1996 年左右，SGI 开始在其提供的编译器中使用新的寄存器约定。这种新约定可以用来建立使用 32 位地址或 64 位地址的程序，分别叫 ″n32″和″n64″。我们暂时不讨论这些，将会在第 10 章详细讨论。

寄存器名约定与使用

*at: 这个寄存器被汇编的一些合成指令使用。如果你要显示的使用这个寄存器(比如在异常处理程序中保存和恢复寄存器)，有一个汇编 directive 可被用来禁止汇编器在 directive 之后再使用 at 寄存器(但是汇编的一些宏指令将因此不能再可用)。

*v0, v1: 用来存放一个子程序(函数)的非浮点运算的结果或返回值。如果这两个寄存器不够存放需要返回的值，编译器将会通过内存来完成。详细细节可见 10.1 节。

*a0-a3: 用来传递子函数调用时前 4 个非浮点参数。在有些情况下，这是不对的。请参 10.1 细节。

* t0-t9: 依照约定，一个子函数可以不用保存并随便的使用这些寄存器。在作表达式计算时，这些寄存器是非常好的暂时变量。编译器/程序员必须注意的是，当调用一个子函数时，这些寄存器中的值有可能被子函数破坏掉。

*s0-s8: 依照约定，子函数必须保证当函数返回时这些寄存器的内容必须恢复到函数调用以前的值，或者在子函数里不用这些寄存器或把它们保存在堆栈上并在函数退出时恢复。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函数调用期间必须保存原来值。

* k0, k1: 被 OS 的异常或中断处理程序使用。被使用后将不会恢复原来的值。因此它们很少在别的地方被使用。

* gp: 如果存在一个全局指针，它将指向运行时决定的，你的静态数据(static data)区域的一个位置。这意味着，利用 gp 作基指针，在 gp 指针 32K 左右的数据存取，系统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：一条是获取有编译器和 loader 决定好的 32 位的地址常量。另外一条是对数据的真正存取。为了使用 gp, 编译器在编译时刻必须知道一个数据是否在 gp 的 64K 范围之内。通常这是不可能的，只能靠猜测。一般的做法是把 small global data (小的全局数据)放在 gp 覆盖的范围内(比如一个变量是 8 字节或更小)，并且让 linker 报警如果小的全局数据仍然太大从而超过 gp 作为一个基指针所能存取的范围。

并不是所有的编译和运行系统支持 gp 的使用。

*sp: 堆栈指针的上下需要显示的通过指令来实现。因此 MIPS 通常只在子函数进入和退出的时刻才调整堆栈的指针。这通过被调用的子函数来实现。sp 通常被调整到这个被调用的子函数需要的堆栈的最低的地方，从而编译器可以通过相对於 sp 的偏移量来存取堆栈上的堆栈变量。详细可参阅 10.1 节堆栈使用。

* fp: fp 的另外的约定名是 s8。如果子函数想要在运行时动态扩展堆栈大小，fp 作为桢指针可以被子函数用来记录堆栈的情况。一些编程语言显示的支持这一点。汇编编程员经常会利用 fp 的这个用法。C 语言的库函数 alloca()就是利用了 fp 来动态调

整堆栈的。

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 fp 被初始化为一个相对与该函数堆栈的一个常量的位置。这种用法对其他函数是不可见的。

* ra: 当调用任何一个子函数时，返回地址存放在 ra 寄存器中，因此通常一个子程序的最后一个指令是 jr ra.

子函数如果还要调用其他的子函数，必须保存 ra 的值，通常通过堆栈。

对於浮点寄存器的用法，也有一个相应的标准的约定。我们将在 7.5 节。在这里，我们已经介绍了 MIPS 引入的寄存器的用法约定。最近在约定方面有一些演化，我们将在 10.8 节中介绍这些变化，比如调用约定的一些新标准。

2.3 整数乘法部件与寄存器

MIPS 体系结构认为整数乘法部件非常重要，需要一个单独的硬件指令。这一点在 RISC 芯片里不多见。一个另外做法是通过标准的整数运算流水线部件来实现一个乘法。这意味着对於每个乘法指令，需要一段软件过程(来模拟一个乘法指令)。早期的 Spacr CPU 就是这样做的。

另外一个用来避免设计一个整数乘法器的做法是通过浮点运算器来实现乘法。Motorola 的 88000 CPU 家族就是提供了这样的解决方案。这样的缺点是损失了 MIPS 浮点运算器是用来做浮点运算的设计初衷。

早期的 MIPS 乘法运算器不是特别快。它的基本功能是将两个寄存器大小的值做一个乘法并将两个寄存器大小的结果存放在乘法部件里。mfhi, mflo 指令用来将结果的两部分分别放入指定的通用寄存器里。

与整数运算结果不一样的是，乘法结果寄存器是互锁的(inter-locked)。试图在乘法结束之前对结果寄存器的读操作将被暂停直到乘法运算结束。

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

MIPS CPU 的整数乘法部件操作相对而言比较慢：乘法需要 5-12 个时钟周期，除法需要 35-80 个时钟周期(与具体 CPU 的实现有关，如操作数的大小)。相对一个同样的双精度浮点运算操作，乘法和除法操作是太慢了。乘/除法并且在内部不是靠流水线来实现的。可见相应的硬件实现是牺牲了速度以换取(指令)简单和节省芯片大小。

汇编器提供了一个合成的乘法指令用来执行乘法并将结果取出放回一个通用寄存器。MIPS 公司的汇编器会通过一系列的移位和加法操作来替换(硬件)的乘法指令， 如果汇编器优化觉得这样更快的话。我对於这一点的意见是优化的工作应该有编译器来

完成，而不是有汇编器来做。

乘法部件不是流水线构造的。每一次只能执行一条指令。上一次的结果将丢失如果下一条乘法指令又开始了，上一次的结果不会象流水线结构那样被写到流水线的 write-back 阶段。(译者注：在流水线方式下，在 write-back 阶段，寄存器-寄存器指令的结果将被写回到结果寄存器)。这一点如果不注意的话，将导致一些非常难理解的问题，导致你的程序的结果不对，比如中断的打扰使得你刚才的乘法结果被冲掉了。

如果一个 mfhi 或 mflo 指令在还没有走到流水线的 write-back 阶段而被中断或异常打断，系统将会重新启动上述读取操作，废掉上一次的读取。但是如果下一条指令是乘法指令并且完成了 ALU 阶段，该乘法指令会与异常处理并行的执行，并有可能覆盖掉 hi 和 ho 寄存器里的内容。那么上述 mfhi 或 mflo 的重新执行将会得到错误的结果。由於这个原因，乘法指令一般不要紧跟在 mfhi/mflo 指令后面，要隔开两条指令(译者着：从而防止 CPU 的指令预取)

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。任何加载或存储机器指令可以写成
lw $1, offset($2)
你可以使用任何寄存器来作为目标和源寄存器。offset 偏移量是一个有符号的 16 位的数字(因此可以是在-32768 与 32767 之间的任何一值)。用来加载的程序地址是源寄存器与偏移量的和所构成的地址。这种寻址方式一般已足够存取一个 C 语言的结构(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)。这种寻址方式实现了一个通过一个常量来索引的数组；并足够使得可以存取堆栈上的函数变量或桢指针；可以提供一个比较合适大小的以 gp 为基址的全局空间以存取静态和外部数据。

汇编器提供一个简单直接存取方式的汇编格式从而可以加载一个在连接时刻才能决定地址的变量的值。

许多更复杂的方式，如双寄存器或可伸缩的索引，都需要多个指令的组合。

2.5 存储器与寄存器的数据类型

MIPS CPU 可以在一个单一操作中存储 1 到 8 个字节。文档中和用来组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
longlong dword 8 "d"代表 ld
int/long word 4 "w"代表 lw
short halfword 2 "h"代表 lh
char byte 1 "b"代表 lb

2.5.1 整数数据类型

byte 和 short 的加载有两种方式。带符号扩展的 lb 和 lh 指令将数据值存放在 32 位寄存器的低位中并剩下的高位用符号位的值来扩充(位 7 如果是一个 byte，位 15 如果是一个 short)。这样就正确地将一个带符号整数放入一个 32 位的带符号的寄存器中。

不带符号指令 lbu 和 lhu 用 0 来扩充数据，将数据存放纵 32 位寄存器的低位中，并将高位用零来填充。

例如，如果一个 byte 字节宽度的存储器地址为 t1，其值为 0xFE(-2 或 254 如果是非符号数)，那么将会在 t2 中放入 0xFFFFFFFE(-2 作为一个符号数)。t3 的值会是 0x000000FE(254 作
为一个非符号数)

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

上述描述是假设一个 32 位的 MIPS CPU。但是 MIPS III 或其上的体系结构实现了 64 位寄存器。可见所有的部分 word 字的加载(包括非符号数)都带符号(包括 0)扩充到高 32 位。这看上去很奇怪但却是很有用的。这将在 2.7.3 节中解释这一点。

这些较小长度的整数扩充到较长的整数的细微区别是由於 C 语言可移植性的历史原因造成的。现代 C 语言标准定义了非常明确的规则来避免可能的二义性。在不能直接作 8 位和 16 位精度的算术的机器中，如 MIPS，编译器对任何包含 short 和 char 变量的表达式中需要插入额外的指令以确保数据该溢出时得溢出：这一点是不希望的，程序效率非常差。当移植一个使用小整数变量的代码到 MIPS CPU 上的时候，你应该考虑找出那些可以安全的转换成整数的变量。

2.5.2 没对齐的加载和存储

MIPS 体系结构中，正常的加载和存储必须对齐。半字(halfwords)必须从 2 个字节的边界加载；字(word)必须从 4 个字节的边界。一个加载没有对齐的地址的加载指令会导致 CPU 进入异常处理。因为 CISC 体系结构，例如 MC680x0 和 Intel 的 x86 确实能够处理非对齐的加载和存储，当移植软件到 MIPS 体系结构时，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负责相应的加载操作从而使得地址对齐的操作对用户程序是透明的。但是这种做法使得程序效率非常慢，除非这样的异常处理非常少。

所有 C 语言的数据类型将严格的按照其数据类型的大小对齐。

当你不知道你要操作的数据是对齐的或者说就是不对齐的，MIPS 体系结构允许通过两条指令来完成这个非对齐的存取(比通过一些列的字节的存取然后移位，加法的效率高得多)。这些代理指令的操作很隐含，比较难以掌握，通常是有宏指令 ulw 的产生的。详细可见 8.4.1 节。

MIPS 另外还提供宏指令 ulh(非对齐的加载半字)。这也是通过合成指令来完成的--两个加载操作，一个移位和一个位或操作。

通常，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(但从一个文件中读取数据或与一个不同的 CPU 共享数据)能够处理非对齐的整数数据是必须的，一些编译器允许你设定一个数据类型是非对齐的，编译器将会产生相应的特殊代码来处理。ANSI 提供#progma align nn，GNU 是通过更简洁 packed 结构属性类型来指定。

即使你的编译器实现了 packd 数据类型，编译器并不保证会使用特殊的 MIPS 指令来实现非对齐的存取。

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中不会经过任何检查--你可以加载一个非法的浮点数据(实际上，你可以加载任意的数据模式)，并不会得到浮点运算错误直到对这些数据进行操作。

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)
```

被扩充为两个连续的寄存器加载：

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

在一个 64 位 CPU 上， l.d 是机器指令 ldc1 的别名。ldc1 完成 64 位数据的加载工作。

任何一个遵循 MIPS/SGI 规则的 C 编译器都将 8byte 的 long(长整数)，双精度浮点变量在 8byte 的地址边界上对齐。32 位硬件不需要这个要求，对齐是为了向上的兼容性：64 位 CPU 如果加载一个没有在 8byte 上对奇的 double 变量，CPU 将进入错误处理，进入异常。

2.6 汇编语言的合成指令

虽然从体系结构的原因我们不能直接用一条指令来完成将一个 32 位的常量取入一个寄存器中， 但是写 MIPS 机器码或许太沉闷了。汇编语言程序员不想每次都得考虑这些。因此 MIPS 公司的汇编器(和其他的 MIPS 汇编器)将会为你合成一些指令。你只需要写一个加载立即数指令，汇编器会知道什么时候通过两条机器指令来实现之。

显然这是很有用的，但是同时自从发明之后也就一直被乱用。许多 MIPS 汇编器通过

将体系结构的特点掩盖起来从而使得不需要合成指令。在本书中，我们将试图尽量少用合成指令，当使用时，会给读者指出来。另外，在下面的指令列表中，我们将会指出合成指令与机器指令的区别。

我的感觉是合成指令是用来帮助程序员的，严肃的编译器应该严格的一对一的产生机器指令代码。但是在这个不尽善尽美的世界里，还是有许多编译器产生合成指令。

汇编器提供的有用的方面包括下列：

* 一个 32 位的立即数加载：你可以在数据码中加载任何数据(包括一个在连接阶段决定的内存地址)，汇编器将会把其拆开成为两个指令，加载这个数据的前半部分和后半部分。

*从一个内存地址加载：你可以从一个内存变量来作一个加载。汇编器通常会将这个变量的高位地址放入一个暂时的寄存器中，然后将这个变量的低位作为一个加载的偏移量。当然这不包括 C 函数里的局部变量。局部变量通常定义在堆栈上或寄存器中。

*对内存变量的快速存取：一些 C 程序包含了许多对 static 和 extern 变量的存取， 对它们加载与存储用 load/store 两条指令开销太大了。一些编译系统避开了这一点，通过一些运行时的支持。在编译的时刻，编译器选择好一些变量(MIPS 公司的汇编器缺省选择那些 8 或更少存储字节的变量)，并将它们放在一起到一个大小不超过 64K 字节的内存区间。运行系统然后初始化一个寄存器--$28，或者说 gp，来指向这个区域的中间位置。
对这些数据的加载和存储可以通过对 gp 寄存器相对位置的一个加载或存储来完成。

*更多类型的跳转条件：汇编器通过对两个寄存器的算术测试来合成一系列的条件跳转。

*简单或不同形式的指令：一元操作，例如，not 和 neg，是通过 nor 或 sub 与永远值是零的寄存器$0 来实现的。你还可以用两个操作数的方式来表示一个三个操作数的指令。汇编器将会把结果存回到第一个指定的寄存器中。

*隐藏跳转延迟槽：在正常的情况下，汇编器将不会让你接触到延迟槽。SGI 汇编器非常灵巧，可以识别指令序列寻找有用的指令并将其放入到延迟槽中。一个汇编 directive .set noreorder 可以用来防止这一点。

*隐含加载延迟：汇编器会检测是否一个指令试图使用一个前面刚加载的数据结果。如果有这样的情况，将会对代码进行移动。在早期的 MIPS CPU 中(没有加载数据互锁)，系统将会插入一个空指令 nop。

*没对齐的移动：不对齐的数据加载和存储指令将会正确地存取半字和字数目，虽然

目标地址是非对齐的。

*其他流水线矫正：一些指令(例如那些使用乘法器的指令)需要额外的限制--例如乘法器的输入寄存器在结果输出之后的第 3 条指令时才能复位并重新使用。你可能不想知道这方面太多的细节，汇编器会替你把补丁填好。

*其他的优化：一些 MIPS 指令(特别是浮点)需要花费很多的指令来产生计算结果，而且在这期间 CPU 是互锁的，因此你不需要考虑这些延迟对你程序正确性的影响。但是 SGI 的汇编器在这方面非常勇敢，会将代码挪来挪去从而提高运行速度。你有可能不喜欢这一点。

纵队，如果你想将汇编源代码(没有用.set noreorder 的代码)与在内存中的指令对应起来，你需要帮助。请使用一个反汇编工具。

2.7 MIPS I 到 MIPS IV: 64 位(和其他)的扩展

MIPS 体系结构自从诞生以来就一直在演变，最为显著的为从 32 位到 64 位。这个扩展非常干净利索，以致在介绍 MIPS 体系结构时我们几乎可以按照 64 位的体系结构来描述，32 位的结构当作是其的子集。本书没有这样做，因为如下几个原因。第一，MIPS 并不是一开始就是 64 位的。如果一开始就按照 64 位来描述，可能会使得你迷惑。第二，MIPS 提供给工业界的一个经验就是一个体系结构如何能够平滑的扩展。第三，本书的材料其实是为 32 位 MIPS 而准备的，当时 MIPS 还没有包含其 64 位扩展。

因此，我们介绍的方法是混合的。通常我们会先介绍 32 位下的□c 能，当介绍到细节的时候，就会既包括 32 位又包括 64 位。在以后我们将用 ISA 作为指令集的缩写。

当 MIPS ISA 演化时，原来 32 位 MIPS CPU (包括 R2000， R3000 和其相应的产品) ISA 都相应的称为 MIPS I。另一个广泛使用的，含有许多重要改进并从而在 R4000 及其后续产品上提供了完整 64 位 ISA 的指令集，我们称之为 MIPS III。

MIPS 的一个优点是，在用户层次(当你在一个工作站上写程序时，你可见的所有代码)，每个 MIPS ISA 都是其前一个的超集，没有任何遗漏，只有增加新的功能。

MIPS II 出现过。但其第一个实现 R6000 马上就被 MIPS III R4000 取代了。除了 MIPS III 的 64 位的整数运算， MIPS II 非常接近于一个 MIPS III 的子集。MIPS II ISA 最近又回来了，随着对 32 位的 MIPS CPU 实现的要求的增加。

如我们已经描述过的，不同的 ISA 层次定义和描述了相应 ISA 层的内容。除去其他内容，这些 ISA 至少定义了在一个保护的操作系统中一个用户程序所要使用的所有的，包含浮点运算的指令。如从指令系统出发，ISA 定义和描述了整数，浮点数和浮点控制寄存器。

每一个 ISA 定义都非常小心的将 CPU 控制寄存器(协处理器 0)，最近将所有的 CPU 控制

寄存器都排除在外。我不知道这有什么帮助，虽然这可以创造更多的 MIPS CPU 咨询业的工作机会，由於 ISA 中隐含了很多信息。例如，如果你想要了解如何对 R5000 的 cache 编程的话，“MIPS IV 指令集”的书是没有任何帮助的。

在实践中，协处理器 0 也伴随这正式的 ISA 一起演化着。与 ISA 的版本类似，协处理器 0 有两个主要的版本：一个是与 R3000(MIPS CPU 中最大家族 MIPS 1 的祖先)，另一个是第一个 MIPS III CPU， R4000。我将称这两个 CPU 家族为 R3000 式的和 R4000 式的。以后的 CPU，如 R5000 何 R10000 都保留了 R4000 式的协处理器构造。

2.7.1 迈向 64 位

1990 年 MIPS R4000 的问世，MIPS 成为第一个 64 位的 RISC 芯片。MIPS III 的指令集提供 64 位的整数寄存器。所有的通用寄存器是 64 位大小的。有一些 CPU 控制寄存器也是 64 位的。另外，所有的操作都产生 64 位的结果，虽然一些从 32 位的指令集继承过来的指令对 64 位的数据没有任何影响。对那些不能兼容的扩展到 64 位来处理 64 位的操作数的 32 位指令，MIPS III 指令集提供了新的增加的指令。

在 MIPS III 中，FPA 有独立的 64 位长的 FP 寄存器，因此你不再需要一对 32 位的寄存器来存放一个双精度的浮点运算值。这个扩展是不兼容的，因此人们可以通过设置一个控制寄存器的模式开关来使得这些寄存器的行为与 MIPS I 一样从而使得旧软件也可以使用。

2.7.2 谁需要 64 位?

到 1996 年，32 位 CPU 已经不能提供足够的地妒 7d 空间给一些大的应用程序。专家们认为程序的大小在指数倍的增长，每 18 个月就翻一番。随着这个增长速度，对地址空间的要求将是每年要增加 3/4 个 bit。真正的 32 位机器(68020, i386)是在 1984 年取代 16/20 位的机器的。因此 32 位机器将会在 2002 年左右变的嫌小了。如果从这个数据让我们觉得 MIPS1991 年的动作太超前了，或许是对的--MIPS 的最大支持者 SGI 直到 1995 年才推出其 64 位的操作系统。

MIPS 技术早期的发展来源于操作系统的研究兴趣，希望通过使用较大的虚拟地址空间从而使得一个对象(object)可以在一段时间内通过其虚拟地址来命名。 MIPS CPU 绝不是在操作系统发展中最有威望的机构。Intel 占据世界市场的 32 位 CPU 等待了 11 年直到 Windows 95 操作系统将 32 位运算带入了巨大的市场。

64 位体系结构的一个特点是计算机可以一次处理更多的位，这可以使得一些要处理大量数据的应用程序，如图形和图像，得到加快。对於多媒体指令的扩充，如 Intel 的 MMX，士不是有必要还不是很明朗。Intel 的 MMX 不仅提供宽广的数据通道，还能满足同时处理在其数据通道上一个字节或 16 位数据。

到了 1996 年，任何一个声称具有长远目标的体系结构都需要相应的 64 位的实现。或许早点实现 64 位计算不是一个坏事。

采用一个平面一维的线性地址空间和将通用寄存器作为指针是 MIPS 体系结构的特点。这意味着 64 位寻址和 64 位寄存器是相伴的。即使不考虑宽的 64 位地妒 7d，增加了宽度的寄存器与 ALU 对一些处理大量数据的程序，如图形或高速通讯程序也是非常有用的。

MIPS 体系结构(和其他一些 RISC 体系结构)带来的一个希望是体系结构朝 64 位的发展使得地址的段式结构(x86 和 PowerPC 体系结构的特点)变得再没有任何必要。

2.7.3 关于 64 位与 CPU 模式转换：数据位於寄存器中

在将一个 CPU 扩充到一个新的领域时，通常“标准”的做法是象很久以前 DEC 公司将其 PDP-11 挪到 VAX 上和 Intel 公司从 80286 升到 i286 和 i386：他们在新的处理器中定义一个模式转换控制，当模式控制启动时，使得处理器运行得象其前代产品一样。

但是模式切换是一种组合起来的一种方法。在一个没有微代码的机器中，这种模式切换是很难实现的。因此 R4000 采用了一种不同的方法：

* 所有的 MIPS II 指令集都保留。
* 只要你仅仅运行 MIPS II 指令，你的程序就是与 MIPS II 处理器是 100%兼容的。每一个 MIPS III 的 64 位寄存器的低 32 位存放着相应的在 MIPS II CPU 时其寄存器的值。

*尽可能的定义 MIPS II 指令，从而使得保持兼容性并且可用在 64 位指令中。

在这里，重要的决定(当你清楚这个问题后，就是一个简单的问题)是，但我们将 64 位 CPU 运行在 32 位兼容状态下时，寄存器高 32 位将存放什么值？有很多种选择，但只有少数几个是简单明了得。

我们可以简单的决定寄存器高 32 位是没定义的。当你将 CPU 运行在 32 位兼容模式下时，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：我们将需要 32 位和 64 位各自的测试和条件转移指令(用来测试寄存器是否相等或通过检查最高位来负数)。

第二种方案相对吸引人一点，当 CPU 运行在 32 位时，寄存器高 32 位保持为 0。这种方法同样要求提供各自的对负数的测试指令和对负数的比较指令。另外，一个 64 位的异或("nor")指令用在两个高 32 位为 0 的值时，不能自然的产生一个高位为 0 的值。

第三种，也是最好的一种方法是将寄存器的高 32 位与第 31 位一样。如果(当仅仅运行 32 位指令时)我们确信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 64 位比较和测试指令与其 32 位的相应指令就都是这个兼容的。所有的位操作逻辑指令也同样(任何对位 31 操作正确的，对位 32 到 63 也同样适用)。

这个正确的方法可以这样来加以描述，将寄存器的低 32 位进行带符号扩展到 64 位。这种方法与寄存器中的值是带符号的还是不带符号的无关。

按照这个方案，MIPS III 需要新的 64 位简单数值计算指令(32 位的 addu 指令，当遇到 32 位溢出时，将会把溢出的结果存放在低 32 位，并将第 31 位扩充至高 32 位---这与 64 位加法是不一样的！)。MIPS III 还需要新的 64 位的存取和移位指令。在需要一个新的 64 位指令时，其指令助记符增加一个“d“，比如 daddu, dsub, dmult 和 ld 等。

略微不是很明显的是 32 位的加载指令 lw。在 64 位下，lw 更精确的意思是加载一个带符号的字(word)，因此一个用在 64 位下的新的指令 lwu 被引入。lwu 意味着高 32 位是用 0 来扩展。
需要增加的指令的数目是由支持现有的 MIPS II CPU 种类的需要和(比如，按照一个常数来移位)支持使用不同的指令操作码(op-code)如何避免在 32 位下固定的只有 5 位的移位数。

所有的 MIPS 指令都详细的列在了第八章。

2.7.4 MIPS III 的其他一些发明
同步 64 位的广泛扩展提供了一个机会来增加一些非常有用的指令(与 64 位数值计算操作无关的)。

多处理器操作

64 位 MIPS 提供了一对指令--加载关联(load linked)和条件存储(store conditional)。它们用来实现软件的 semaphore，可用在共享内存的多处理器系统中。它们的功能与最近的 CISC 体系结构提供的原子性的 RMW(读-改-写)指令和锁指令是一样的。但是，RMW 和锁指令在一个大的多处理器系统中效率是不好的。我们将在 5.8.4 节中解释加载关联和条件存储的操作。在这里，下面是对它们功能的一些介绍。

ll 是一个普通的加载一个 word 的指令，但是它在一个特殊的内部寄存器中保持这个地址的记录。sc 是一个存储一个 word 的指令，但是它只在如下条件下才存储：

* 自从上次在这同样地址上的 ll 指令之后，CPU 没有发生任何中断或异常，并且

* (对多处理器系统)，没有别的 CPU 发出写操作或试图一个写操作并且写的地址包括了 ll 指令使用的地址。

sc 指令会返回一个值来告诉程序存储是否成功。

虽然 ll 和 sc 指令是为多处理器系统设计的，也可以被用在单处理器系统上。从而可以实现一个 semaphore 而不需要关闭中断。

封闭循环转移(可能循环)

高效的 MIPS 代码要求编译器能够在大多数延迟槽上安排有用的工作。在许多情况下，逻辑上在跳转指令之前的那条指令是合适的选择。显然，如果这个跳转指令是一个条件跳转并且在其之前的那个指令是计算这个跳转条件的，那么就不能把其之前的

那条指令放入延迟槽中。

这种情况在包含一个循环的跳转中经常出现。循环越小，编译器就越难找到一个之前的指令并方入延迟槽中。

在一个循环里面，编译器的第二种选择是在延迟槽中存放一个跳转指令的目的地的那条指令的备份。并且将跳转目标地址提高一个 word。这个调整不会使得程序变小，但确实能使程序运行加快。但是这个方法通常是不可能的。当一个循环结束时，在延迟槽里的指令将会被执行，这使得编译器很难判断这个行为是否会造成任何损害。

在这里编译器需要的是一个只有在跳转被执行时延迟槽里的指令才被执行的跳转指令。这是 MIPS III 指令集可以提供的功能。这些指令称之为“可能跳转”(branch likely)--这个命名非常容易迷惑人。它们的助记符是在清 b 有的指令助记符后面加一个"l"：因此 beq 产生 begl 指令。其他依此类推。

条件异常

随着 MIPS III， 提供了一系列指令可以依据一个条件来使 CPU 进入异常处理：测试条件与“set if ..."指令是一样的。这些指令在 C 语言中没有相应的语法，但是可以用来实现那种动态检查数组越界的编程语言。

扩充的浮点数

R6000 将浮点寄存器扩充到了 64 位宽度。但是我把其当作是 MIPS III 扩充到 64 位的一部分。如果新的 MIPS II 有浮点处理器(不太可能)，一般而言是 32 位的。

2.8 基本地址空间

相对於其他 CISC CPU， MIPS 处理器对地址空间的使用有些细微的不同。这一点有时会使人迷惑。请仔细阅读这一节的第一部分。我们将先介绍 32 位 CPU 的情况，然后再介绍 64 位。耐心点你将会在以后知道我为什么这样做。

下面是一些概述。在 MIPS CPU 里，你的程序中的地址不一定是芯片真正访问的物理地址。我们分别称之为：□
’7b 序地址和物理地址。

一个 MIPS CPU 可以运行在两种优先级别上， 用户态和核心态。MIPS CPU 从核心态到用户态的变化并不是 CPU 工作不一样，而是对於有些操作认为是非法的。在用户态，任何一个程序地址的首位是 1 的话，这个地址是非法的，对其存取将会导致异常处理。另外，在用户态下，一些特殊的指令将会导致 CPU 进入异常状态。

在 32 位下，程序地址空间划分为 4 个大区域。每个区域有一个传统的名字。对於在这些区域的地址，各自有不同的属性：

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 MMU 作转换。除非 MMU 的设置被建立好，这 2G 地址是不可用的。

对於没有 MMU 的机器，存取这 2G 地址的后依具体机器相关。你的 CPU 具体厂商提供的手册将会告诉你关于这方面的信息。如果想要你的代码在有或没有 MMU 的 MIPS 处理器之间有兼容性，尽量避免这块区域的存取。

kseg0: 0x8000 0000 - 0x9FFF FFFF(512M): 这些地址映射到物理地址简单的通过把最高位清零，然后把它们映射到物理地址低段 512M(0x0000 0000 - 0x1FFF FFFF)。因为这种映射是很简单的，通常称之为“非转换的“地址区域。

几乎全部的对这段地址的存取都会通过快速缓存(cache)。因此在 cache 设置好之前，不能随便使用这段地址。通常一个没有 MMU 的系统会使用这段地址作为其绝大多数程序和数据的存放位置。对於有 MMU 的系统，操作系统核心会存放在这个区域。

kseg1: 0xA000 0000 - 0xBFFF FFFF(512M): 这些地址通过把最高 3 位清零的方法来映射到相应的物理地址上，与 kseg0 映射的物理地址一样。但 kseg1 是非 cache 存取的。

kseg1 是唯一的在系统重启时能正常工作的地址空间。这也是为什么重新启动时的入口向量是 0xBFC0 0000。这个向量相应的物理地址是 0x1FC0 0000。

你将使用这段地址空间去存取你的初始化 ROM。大多数人在这段空间使用 I/O 寄存器。如果你的硬件工程师要把这段地址空间映射到非低段 512M 空间，你得劝说他。

kseg2: 0xC000 0000 - 0xFFFF FFFF (1G): 这段地址空间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，不能存取这段区域。除非你在写一个真正的操作系统，一般来说你不需要使用这段地址空间。

2.8.1 简单系统的寻址

MIPS 的程序地址很少与真正的物理地址一致。但对於简单的嵌入式软件而言可以用 kseg0 和 kesg1 这两段地妒 7d 空间。它们朝物理地址的映射关系是非常直接了当的。

从 0x20000 0000 开始的 512M 物理地址空间在上述 kseg0, kseg1 和 kseg2 中没有任何的映射。你可以通过设置 MMU TLB 的方式来访问，或者使用 64 位 CPU 的一些其他额外的空间。

2.8.2 核心与用户权限

在核心态下(CPU 启动时)，PU 可以作任何事情。在用户态下，2G 之上的地址空间是非

法的。任何存取将会导致系统异常处理。注意的是，如果一个 CPU 有 MMU，这意味着所有的用户地址在真正访问到物理地址之前必须经过 MMU 的转换,从而使得 OS 可以防止用户程序随便乱用。对於一个没有内存映射的 OS，MIPS CPU 的用户态其实是多余的。

另外，在用户态下，一个指令，特别是那些 CPU 控制指令，是不能使用的。

要提及的是，当你作核心态和用户态切换时，并不意味着□c 能的改变，只不过是意味着某些功能在用户态下不能使用了。在核心态下，与用户态一样，CPU 可以存取低段地址空间。这个存取也是通过 MMU 的转换。这一点与用户态下一样。

另外要注意的是，虽然如果把操作系统运行在核心态下，平常的代码运行在用户态下是一种不错的选择。但如果反之也不为过。有些系统，包括□c 多实时操作系统，都是全部运行在核心态下。

2.8.3 64 位 CPU 的地址空间

MIPS 地址的形成是通过一个 16 位的偏移量和一个寄存器。在 MIPS III 或更高版本的 CPU 里，一个寄存器是 64 位。因此一个程序地址是 64 位的。这样大的地址空间允许我们耐心的将其划分。请参阅图 2.2。

首先要注意的是 64 位内存映象是包含在 32 位内存映象里面的。这是个有点奇怪的方法，就象 Dr. who 的“Tardis”--里面比外面要大的多。这一点是通过 2.7.3 节介绍的规则来实现的：当模拟一个 32 位指令集的适合，寄存器存放的是其 32 位的带符合位扩展的 64 位值。因此，一个 32 位程序存取的是 64 位程序空间的最低和最高的 2G。换句话说，64 位 CPU 的地址空间的最低和最高区域是和 32 位情况下一样的，64 位扩展的地址部分在这两者之间。

在实践中，扩展的用户地址空间和超级用户权限的地址空间一般而言没有太大的用处，除非你在写一个虚拟内存操作系统。因此许多 MIPS III 的使用者仍然定义 32 位的指针。64 位下那些大块的不需要 MMU 转换的窗口可以克服 kseg0 和 kseg1 512M 的局限，但是我们可以通过对 MMU 编程来同样达到这一点。

2.8.4 流水线 hazard

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一个延迟。体系结构的设计者要决定这些延迟中的哪一些对於编程员是可见的。将时序 上的缺点隐含起来使得程序员的编程模型简单，比如，CPU 究竟在干什么。当然与此同时，这将对硬件实现引入复杂性。将调度问题留给程序员和其软件工具将简化硬件部分，但同时产生编程和移植的问题。

正如我们已经提过几次，MIPS 体系结构使得其一些缺点/特点是可见的。程序员和编译器要负责配合 CPU 使得其正常工作。下面一些是关于流水线的方面：

* 跳转延迟：在所有的MIPS CPU里，紧跟着跳转指令的指令(在延迟槽中)会被CPU执行，即使跳转成左5c。在MIPS II指令集中引入的“可能跳转”(branch-likely)指令中，在延迟槽中的指令只会在跳转被接受的情况下被执行。详细可见8.4.4关于”可能跳转“的基本原理。程序员或编译器必须找到一个有用的，至少是无害的指令放在延迟槽中。但是，除非你指定，汇编器将会使得跳转延迟是透明不可见的。

*加载延迟：在MIPS I指令集里，load指令后面的指令(在加载延迟槽)不能使用刚用load加载的数据。一个有用的或无害的指令需要放在加载延迟槽里来将数据加载和数据使用分开。与跳转延迟一样，除非你指定，汇编器将会使得这个延迟处理对你是透明不可见的。

*整数乘法/除法问题：整数乘法部件是和ALU部件分开的，没有实现“精确异常”(请参阅5.1节关于精确异常的定义)。解决方法很简单，通常是通过汇编器--在读取上一个乘除法的结果值之后，你需要避免立刻启动下一个乘除法运算。为什么这个解决方法是必须的和足够的很负责(请参阅5.1节)。

*浮点数(协处理器1)的缺点：任何一个浮点运算几乎都要花费多个CPU时钟周期来完成，MIPS FPA通常有多个独立的流水线部件。在这种情况下，硬件可以把流水线隐含起来；FP计算可以与其后的指令并行的执行。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就会停止下来。编译器需要大量的优化工作在这方面，比如重复指令比率表，各种目标CPU的延迟表等。当然，你没必要依赖这些来使得你的程序工作。

如果一个浮点计算没有流水线hazard，并不意味着浮点运算协处理器与整数运算部件的交互没有流水线hazard。这里面有两方面原因。

第一，从浮点运算器移动数据到整数寄存器的指令--mfc1，传送数据的时刻是在下一个时钟周期，与“load”具有同样的时序要求。就象load一样，在MIPS 1 CPU中，这是个hazard，但在后来的硬件中，被利用硬件的内置锁(interlock)解决了。优化的编译器会利用延迟槽完成一些有用的工作。

第二，测试一个浮点运算的条件的指令不能直接跟在产生那个条件的浮点比较操作后面。对大多数MIPS CPU实现，需要一个指令的延迟。

* CPU控制指令问题：这个部分非常容易迷惑人。当你改变CPU状态寄存器的内容时，你潜在地在影响发生在流水线所有阶段的东西。因为关于CPU控制系统的结构描述是与具体的实现有关，因此没有ISA指令集方面的规则可以遵循。遗憾的是CPU厂商至今没有提供有关相应的文档。

请参阅第三章关于MIPS CPU控制指令的总结，然后请阅读附录A关于R4000 CPU的时序问题。

第四章 Cache for MIPS

没有Cache的MIPSCPU不能称为真正的RISC。可能这样说不公平。但为了一些特殊的目的，你可以设计一个含有小而紧密内存的 MIPSCPU，而这些内存只需要固定个数的流水线步骤

（最好是一个）就可以被访问到。但绝大部分 MIPS CPU 都是含有 cache 的。
这一章将介绍MIPS的cache怎样工作和软件应该怎么做才能使它可以被使用而且是可靠的。MIPSCPU 重新启动后，cache 的状态是不确定的，所以软件必须非常小心。你有一些线索知道 cache 的大小（如果你直接知道 cache 的大小后去初始化，这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 cache 和获取特殊入口。
对于实时应用程序的程序员，希望在 CPU 运行时能够正确地控制 cache。我们也将讨论怎么做，虽然我对使用一些窍门方式有怀疑。
当然这些也随着 MIPSCPU 的发展而进步。对于早期的 32 位 MIPS 处理器，初始化 cache 或者使其无效，首先让 cache 进入一种特殊的状态，然后通过普通的读写操作来完成。对于后来的处理器，一些特殊的指令被定义出来做这些相关的操作。

4.1 cache 和 cache 的管理

cache 的工作就是将内存中的一部分数据在 cache 中保留一个备份，使这些数据能一个固定的极短的时间内被快速的存取并返回给 CPU，这样能保证流水线的连续运行。
绝大部分 MIPSCPU 针对指令和数据有其各自的 cache（分别称为 Icache 和 Dcache)，这样读一条指令和一个数据的读操作或者写操作就能同时发生。
老的 CPU 家族（象 x86)为了保证被写入 CPU 的代码的一致性，所以没有 cache。现在的 x86 芯片拥有更灵活的硬件设计，从而保证软件没有必要从更本上了解 cache（如果你正在装一台机器跑 MS/DOS，它将在本质上提供一致性）。
但因为 MIPS 机器有各自的 cache，所以就没有必要那么灵活。cache 对于应用程序来说必须是透明的，除了除了能感觉到运行速度的增加。但对于系统程序或者驱动程序，拥有 cache 的 MIPSCPU 并没有尝试 cache 对它们也是透明的。cache 仅仅使 CPU 跑得更快，而不能给系统程序员有所帮助。在象 Unix 一类的操作系统中，操作系统能对应用程序完全隐藏 cache，当然对于更多不能的胜任的操作系统，其也能很好的隐藏大部分 cache 的处理，但你可能必须知道在什么时候需要调用适当的子程序来对 cache 做一些必要操作。

4.2 cache 怎样工作

从概念上讲，cache 是一个相连内存（associative memory），当数据被写入时用数据的一部分作为关键字来标志的一块存储区域。在 cache 中，关键字是整个内存的地址。提供一个相同的关键字给相连内存，你将得到相同的数据。一个真实的相连内存在存入条目时，将完全按照它们的关键字，除非它已经满了。然而，由于需要这个当前的关键字必须和所有被存的关键字同时比较,因此任何大小的真实相连内存不是效率低或速度慢,或者就是两者都有。怎样我们才能设计有用的高速缓存,使其不仅效率高而且速度快呢？图 4.1 展示了一种最简单高速缓存的基本设计方案，直接映射（direct-mapped）高速缓存。它被 1992 年以前的 MIPSCPU 广泛使用。
直接映射 cache 由许多块简单的高速缓存排列构成（通常每一块称之为一 line），通过地址低位在整个范围内做索引。cache 的每一条 line 都包含一个字或者几个字的数据和一个标签（tag）区域，tag 记录着数据所在内存的地址。
当一个读操作时，每一条 line 都可以被访问到，tag 将和内存地址的高位做比较；如果匹配的话，我们知道是找到正确的数据了，这被称之为命中（hit）。如果在这一块中有超过一个字的数据，对应的那个字的数据通过地址的最低几位来选择出来。
如果 tag 没有匹配，这称之为没有命中（miss），那么数据需要从内存中读入，然后复制到 cache 对应的 line 中。这对应 line 中原来的数据将会被抛弃，如果 CPU 又需要被抛弃的数据时，需要再次从内存中取得。

这样的直接映射 cache 有一个特征，就是对于任何一个内存地址，在高速缓存中只有唯一的一条 line 可以用来保存其数据。这样有好处也有坏处。好处就是这样的架构简单，可以使 CPU 跑得更快。但简单也有其不好的一面：如果你的程序要不停地交替使用两个数据，而它们刚好要对应高速缓存中的同一块（可能是它们对应内存地址的低位刚好一样），这样这两个数据就会不停的将对方替换出高速缓存，以至高速缓存的效率被彻底的降下来。
而真正的相连内存将不会遇到这样的折腾，但对于任何合理大小，它将是难以想象的复杂、昂贵和速度缓慢。
折衷的办法就是使用 two-way set-associative cache，其实就是两个 direct-mapped cache 并联，在它们中同时匹配内存位置。如图 4.2。这时对应一个地址将有两次机会命中。Four-way set-associative cache （就是有四个直接映射的子高速缓存)在 cache 的设计中也是很平常的。但是这是有惩罚的。一个 set-associate cache 比起直接映射 cache 来需要更多的总线连接，所以 cache 太大以至于很难在一块芯片上构造直接映射。
不过也有巧妙的地方，由于直接映射 cache 对于你需要的数据只有唯一的候选者，所以把一些东西放到 tag 匹配前运行是可能的（只要 CPU 不做和着个数据有关的操作)。这样可以提高每一个时钟利用率。
由于当运行一段时间后 cache 会被装满，所以当再次存放从内存读来的数据时，就会抛弃一些 cache 内原有的数据。如果你知道这些数据在 cache 和内存中是一致的，那么你可以直接把 cache 中的备份抛弃；但如果 cache 中的数据更新的话，你就需要首先把这些数据存回到内存中。
这就给我们带来一个问题，cache 怎样处理写操作？

4.3 Write-Through Caches in Early MIPS CPUs
CPU 不能仅仅是读数据（就象上面的讨论)，它们也要写数据。由于 cache 只是将主存中的一部分数据做一个备份，所以有一个显而易见的方法来处理 CPU 的写操作，被称之为 Write-Through cache。
对于 Write-Through cache，写操作时 CPU 总是将数据直接写到主存中去；如果对应主存位置的数据在 cache 中有一个备份，那么 cache 中的那个备份也要被更新。如果我们总是这样做的，那么 cache 中的任何数据将和主存中的保持一致，所以只要我们需要我们就可以抛弃任何一条 cahce line 的数据，并且除了消耗时间不会丢失任何东西。
当然这也是有危险的，当我们让处理器等待写操作结束时，处理器的运行速度将彻底的降下来，不过我们能修复这个问题。可以将要写入主存的数据及其地址先保存在另一边，然后有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内容的地方被称之为写操作缓冲区 （write buffer)，它是先入先出的（FIFO)。
早期的 MIPS CPU 有一个直接映射的 write-through cache 和一个写操作缓冲区，还有一个 R3000 的激发设置。它在同一芯片上构造 cache 控制器，但需要额外的高速存贮器芯片来存贮 tag 和数据。只有 CPU 跑一些特殊的程序很平均地产生的写操作，主存系统在这种工作方式下才能很好的消化这些写操作并工作的很好。
但 CPU 运行速度的增长比存贮器块得多。某些时候当 32 位的 MIPS 让位给 64 位 R4000 后，MIPS 的速度就已经超过存贮器系统可以合理消化所有写操作的临界点了。

4.4 Write-Bach Cache in Recent MIPS CPUs
早期的 MIPS CPU 使用简单的 write-through cache。后来的 MIPS CPU 由于速度太快而不能适用这种方法，它们会陷入存储系统的写操作中，速度慢得像爬行。
解决的方法就是把要写的数据保留在 cache 中。要写的数据只写到 cache 中，并且对应的那

条 cahce line 要做一个标记，使我们肯定不会忘记在某个时候把它回写到内存中(一条 line 需要回写，称之为 dirty)。
Write-back cache 还可以分成几种不同的子处理方式。如果当前 cache 中没有要写地址所对应的数据，我们可以直接写到主存中而不管 cache，或者可以用特殊的方式把数据读入 cache，然后再直接写 cache，后面这种方式被称之为写分配（write allocate)。用一种自私的观点来看一个程序运行在一个 CPU 上，写分配（write-allocate）看起来象浪费时间；但是它可以使整个系统的设计变得简单，因为在程序运行时读写内存都读或者写都是以一条 cache line 大小为单位的块进行操作。
从 MIPS R4000 开始，MIPS CPU 在芯片内拥有 cache，而且都支持 write-through 和 write-allocate 两种工作模式，line 的大小也是支持 16byte 和 32byte 两种。
MIPS cache 的这些工作模式可以被应用到使用 sillicon Graphics 设计 R4000 和其他大型 CPU，其他计算机系统也因为多处理器系统而被这些 cache 工作模式影响到。

4.5 Cache 设计的其他选择
在上个世纪八十和九十年代针对怎样设计 cache，做了很多工作和研究。所以下面还有许多其它的设计选择。
Physically addressed/virtually addressed:
当 CPU 在运行成熟的操作系统时，数据和指令在程序中的地址（程序地址或虚拟地址）会被转换成系统内存使用的物理地址。
如果 cache 纯粹地在物理地址方式下工作，将很容易被管理（我们将在后面讨论为什么)。但合法的虚拟地址可以让 cache 更早地开始查询匹配工作，这样可以使系统跑的稍微块一点。
但虚拟地址有什么问题呢？它们不是唯一的；当许多不同的程序在 CPU 不同的地址空间中运行，它们可能会共享同样的虚拟地址而使用不同的数据。当我们切换不同的地址空间时，每次都需要重新初始化 cache；这种方式在很多年前被使用，可以作为针对非常小的 cache 的一种合理解决方法。但针对大的 cahce 这种方式不仅可笑而且效率低下，我们需要一块区域来辨别 cache tag 中的地址空间，以至我们不被它们混淆。
这儿还有其它关于虚拟地址更细致的问题：相同的物理地址可以在不同的任务中被不同的虚拟地址描述。这就会导致相同物理地址的内容会被映射到不同的 cache 条目中（因为它们对应不同的虚拟地址，所以会被不同的索引所选中)。这样的情况必须被操作系统的内存管理所避免掉。详细的情况将在 4.14.2 节介绍。
从 R4000 起，MIPS 的主 cache 都使用虚拟地址索引，从而提供快速的 cache 索引。但对于作为标记符来标记每一个 cache-line，物理地址比虚拟地址更好。物理地址是唯一的而且效率更高，因为这样的设计显示出 CPU 在做 cache 索引的同时可以把虚拟地址转换成物理地址。

line 大小的选择（Choice of line size）：
line 的大小是对应每一个 tag 可以存贮多少字的数据。早期的 MIPS 的 cache 对应一个 tag 只能存贮一个字的数据。但对应一个 tag 能存贮多个字的数据更好，尤其是内存系统支持快速的 burst read。现代的 MIPS cache 趋向于使用四个或者八个字大小的 line，并且更大的第二层和第三层 cache 使用更大的 line。
当 cache miss 发生时，整个一条 line 的数据都要从内存中获得。但很可能会取来几 line 的数据；一个字的 cache line 的 MIPS CPU 经常是一次就取多个字的数据。

分开/统一（Split/unified）：
MIPS 的主 cache 总是分成 I-cache 和 D-cache，取指令时察看 I-cache，读写数据时察看 D-cache。（顺便说一下，如果你想执行 CPU 刚刚拷贝到内存的代码，你必须不仅仅要是 D-cache 一部分无效使这些代码数据在 D-cache 中不再存在，而且还要保证它们被装入 I-cache）
但是不在同一块芯片上的第二层 cache 很少也按这种方式来分成两块。这样就没有什么真的优势可言了。除非你能针对两种 cache 提供分开的数据总线，但这又会需要太多的管脚。

4.6 Cache 管理（Magaging Caches）
Cache 系统在系统软件的帮助下，必须保证任何应用程序数据的一致性，和它们在没有 cache 的系统下一样，尤其是 DMA I/O 控制器（直接从内存中取得数据）取得程序认为已经写过的数据。
对于 CISC CPU，通常都不需要系统软件对 cache 的帮助；因为它会花费额外的内存空间、silicon area、时钟周期来使得 cache 变得真正的透明。
在系统启动的时候 MIPS CPU 需要初始化它的 cache；这是一个十分复杂的过程，下面有关于它的几点建议。但当系统启动后运行到三种情况 CPU 必须加以干涉。

.在 DMA 设备从内存取数据之前:
如果一个设备从内存中取得数据，它必须取得正确的数据。如果 D-cache 是 write-back，并且程序已经写了一些数据，那么很可能其中一些正确的数据还保留在 D-cache 中而没有写回到主存中去。CPU 当然不可能看到这个问题；如果 CPU 需要这些数据，它会从 cache 中得到正确的数据。
所以在 DMA 设备开始从内存中读数据前，任何一个将被读数据如果还保留在 D-cache 中，必须被写回到内存中。

. DMA 设备写数据到内存:
如果一个设备要将数据存贮到内存中，要使 cache 中任何对应将要写入内存位置的 line 都无效化，这是非常重要的。否则，CPU 读这些位置的数据，将得到错误的数据。cache 应该在数据通过 DMA 写入内存之前将对应的 cache line 无效化。

. 拷贝指令:
当 CPU 自己为了后面的执行而写一部分指令到内存中，你首先必须保证这些指令会被回写到内存中，其次保证 I-cache 中对应这些指令的 line 会被无效化。在 MIPS CPU 中，D-cache 和 I-cache 是没有任何联系的。（当 CPU 自己写指令到内存中时，这时候指令是被当作数据写的，很可能只被写到 cache 中，所以我们必须保证这些指令都会被回写到内存中；为什么要使 I-cache 无效化，这和数据通过 DMA 直接写入内存中要无效 cache 一样的原因。）

如果你的软件需要解决这些问题，就需要针对 cache line 的两个独特的操作。
第一个操作被称之为回写操作。CPU 必须能够针对地址在 cache 中查找对应的 cache line。如果找到，并且对应 line 是 dirty，就需要把这条 line 的数据写回到内存中。
CPU 增加了其他不同层次的 cache（速度和大小），来减少 miss 的处理。所以设计者可以使内层的 cache 机构简单，从而使它能在很高的时钟频率上作查询。这样很显然越往内层的

cache 就会越小。从 1998 年开始，许多高速的 cpu 都在同一块芯片上采用第二级 cache，主 cache 的大小变小，双重 16K 的主 cache 受到青睐。

不在同一块芯片上的 cache 通常都是直接映射的，因为组相连的 cache 系统需要更多的总线从而需要更多的管脚来连接。这还是一个值得研究的领域；MIPS R10000 采用只有一个数据总线的二路组相连 cache，如果命中的不是希望的那一组，通过一段延时后在返回数据来实现（两个组共用一个数据总线）。
在 cache 的发展过程中，产生了两类主要的软件接口来针对 cache。从软件的观点来看，一类是建立在以 R3000 为代表的 32 位 MIPS CPU 的基础上；另一类是建立在以 R4000 为代表的 64 位 MIPS CPU 上的。R3000 这一类型的 MIPS CPU 的 cache 是 write-through，直接映射的，物理地址为索引。cache 访问的最小单位是一个字，所以写一个字节（或者是写小于一个字）的操作必须被特殊的处理。在读写这一类数据是 cache 管理采用特殊的模式。

为什么不通过硬件来管理 cache?
通过硬件来管理 cache 通常被称为“爱管闲事”。当另一个 cpu 或者是 DMA 设备访问内存时，被访问地址对应的内容对于 cache 来说是可以看到的。

4.7 第二层和第三层 cache
在大型的系统中，通常需要一个嵌套的多层 cache。一个小而快的主 cache 最接近 cpu。访问主 cache 出现 miss 时，不是直接从内存中查找而是从第二层 cache 中查找。第二层 cache 在速度和大小上是介于主 cache 和内存之间。cache 层次的数目可以通过内存速度和 cpu 最快访问速度比较来决定；由于 cpu 速度发展比内存的发展快得多，在过去的 12 年里桌上型电脑系统从没有 cache 发展到有两层 cache。九十年代后期的最快 cpu 速度大约可以达到 500MHz，拥有三层 cache。

4.8 MIPS CPU cache 的构造
通过观察 cache 采用模式和层次的发展（看表 4.1），我们可以将 MIPS CPU 分成两类，古老的和现代的。
当时钟的速度变得越快，我们就能看到越多得 cache 构造，因为设计者为了应付 CPU 跑得速度比内存系统越来越快。为了保证运行的顺畅，cache 必须提高运行速度，保证提供数据的速度比外围得存贮器要快，同时也要保证尽可能多命中。相比较 R4000 类型的 CPU，主 cache 是 write back 类型，是 write allocate ,virtually indexed,physically tagged, 二路或四路组相连的 cache。
许多 R4x00 和其后续 cpu 在同一块上拥有第二层 cache 的控制器，1998 年出现了这样的第一块 cpu。
由于两种产生的不同，我们将分两节来详细介绍。

注意！一些系统的第二层 cache 不是由 mips cpu 内部的硬件来控制的，而是建立在内存的总线上。对于这类 cache 的软件接口将具有系统特殊性，和象这章介绍的由 cpu 内部控制的 cache 的软件接口相比，可能有很大的不同。

4.9 Programming R3000-Style Caches
MIPS R2000 打破了芯片内 cache 控制器的基础，将 cache 额外的分成 I-cache 和 D-cache。这是一个后见之明，不会让人感到惊讶，就是这样一个先驱者的冒险导致了后面很多事端。

cache 有一个特殊的软件访问缺点。
为了节省芯片管脚，cache 将不能拥有不同的闸门来执行字节、半个字和其他小于一个字单位的写操作。所以在 R2000 系列中对 cache 执行一个小于字单位的写操作时，会回写到主存中，并将 cache 中这个字所在的 Line 无效化。这样针对 cache 管理，提供了一个使 cache 无效的方法：只用写一个字节就行了。
你可以看到支持这些简化的观点。R2000 设计者提出理由小于字的操作通常用于字符操作，字符操作总是由库函数提供，而这些库函数用整个字的操作来重写。这些假设总是被认为对对错错，或者半对半错。
直到认识到不是所有系统都能用相同的函数库，而且每个字节写操作都使所在 cache 无效也不是一个好主意，这些争论才没有继续下去。因为这是不能被容忍的，所以出现了一个很大的改动，R3000 系列的 cpu 通过一个 RMW(read-modify-write)序列来执行小于字单位的写操作。这个 RMW 出现在所以的 32 位的 mips cpu 中，并增加了一个 时钟周期来作为这样一个写操作的延时。
这样 cache 无效的机制被带入困境；R2000 因为它的奇怪习惯而有一个优点，可以通过字节的写操作来使 cache 无效化。而 R3000 cache 需要用一个叫 isolation 的模式来挽救，原来这种模式只是用于 cache 诊断的。RMW 队列因为这种模式而受到压制，在那种状态下小于一个字单位的写操作还是会让该字所处的 line 无效化。这是不幸的但不是悲惨（灾难）的，对于一些运行着的系统做一些事有着更有益的地方。显著的就是当 cache 在 isolation 模式时的时候，cache 将没有读写操作，任何读写操作将直接和内存打交道。

4.9.1 Using Cache Isolation and Swapping
所有的 R3000 系列 cpu 的 cache 都是 write-through 模式的，这就是说 cache 中不会拥有比内存中更新的数据。也就是说 cache 中的数据从来都不需要回写到内存，所以我们只需要能使 D-cache 和 I-cache 无效就行了。
只需要不同的 cache 操作按照内存顺序来做 cahce 的管理，并且 cache 的管理没有必要通过特殊的内存地址空间。所以这儿有一个状态寄存器有一个 SR 位能够使 D-cache 关闭 isolation 模式；在这种模式下读写操作只影响着 cache，读还是会命中但不管 tag 是不是相等。当 D-cache 处于 isolation 模式时，小于一个字单位的写操作会使对应 cache Line 被无效化。

CAUTION!!!
当 D-cache 处于 isolation 模式，任何读写操做不会受其对应地址或 TLB 条目的影响而按照非 cache 的情况操作。这样的结果就是 cache 管理程序必须保证有些数据是不可以被访问的；如果你能通过你的编译器做到很好的控制，并且能过保证所有你用的变量都保存在寄存器中，你才能在很高级别的语言中写它们。还必须保证运行这些程序时屏蔽中断。

I-cache 在通常运行模式下也是完全不可访问的。所以 CPU 提供了另一种模式，cache 交换(swapped)，通过设置状态寄存器的 SwC 位；这时 D-cache 可以担当 I-cache，I-cache 可以担当 D-cache。当 cache 是交换模式时，isolated 的 I-cache 条目可以被读、写和无效化。D-cache 可以完美的充当 I-cache 使用（可能 I-cache 也可以通过初始化使之象 D-cache 一样工作），但 I-cache 不能完全的充当 D-cache。这也是靠不住的，当 cache 是交换模式时有用，isolation 却没有用。
如果你需要使用交换的 I-cache 来存储字单位的数据(和以前一样小于字单位的数据写操作会使该数据对应的 line 被无效化)，你必须保证在返回到正常模式时对应的 cache line 必

须被无效化。

4.9.2 Initializing and Sizing 初始化和判断大小

当机器启动时 cache 的状态是不确定的，所以这时读 cache 结果也是不可预知的。你也应该认识到机器重起后状态寄存器的 SwC 位和 IsC 位也是不确定的，所以在对 cache 读写前（即使在非 cache 的情况）启动软件最好能将这些状态设为可知的。

不同的 MIPS CPU，cache 有不同的大小。为了保证你软件的可移植性，最好能在初始化的时候计算出 D-cache 和 I-cache 的大小。这样比直接配置一个给定的值好。

下面将介绍怎样获得 cache 大小的值：

a. Isolated cache，让 I-cache 处于交换模式。

b. 在 R3000 系列 CPU 中，cache 的大小可能是 256K，128K，64K，32K，16K，8K，4K ，2K，1K 和 0.5K（K 等于 1024，单位是字节)。将这些可能的值 n（上面那些值中的一个）写到物理地址等于它们本身的地方（有大到小）。最简单产生物理地址是用 Kseg0 段地址（n+0x80000000)。因为 cache 地址是重叠循环的，那么如果 n 是 cache 大小的倍数，那么它就会被后面小的值所覆盖。

c. 所以读物理地址零（也就是 0x80000000），就能得到 cache 大小的值。

初始化 cache，你必须保证每一个 cache 条目都被无效化，而且正确对应一个内存位置，所含的之值也是正确的：

a. 检查状态寄存器 SR 的 PZ 位是不是位零（为 1 的话，关闭奇偶位，对于同一个芯片上的 cache 这不是一个好主意)。

b. isolated D-cache，并使它和 I-cache 交换。

c. 对于 cache 的每一个字，先写一个字的值（使 cache 的每条 line 的 tag、数据、和奇偶位都正确)，然后再写一个字节（使每条 line 都无效)。

不过要注意当对于每条 line 有四个字的 I-cache，这样做效率就很低；因为只要写一个字节就足够使每条 line 无效了。当然除非你要经常调用这个使 cache 无效程序，否则这个问题是不会表现的很明显。不过如果你想根据实际情况来优化 cache 无效化程序，就需要在启动的时候确定 cache 的结构。

4.9.3 cache 无效化(Invalidation)

使 cache 无效，按照下面的流程：

a. 计算出你使 cache 失效所需的地址范围。使用超过 cache 大小的范围是浪费时间。

b. 使 D-cache 孤立。一旦被孤立后你就不能读写内存，所以你必须花费所有的代价来防止异常的产生。关闭所有的中断并保证后面的程序都不会导致内存访问异常。

c. 如果你还想使 I-cache 失效，使 cache 处于交换模式。

d. 在刚才计算出的地址范围内针对每一条 line 写一个字节的内容。

e. 关闭 cache 的交换和孤立模式。

通常你应该在 I-cache 打开的模式下运行使 cache 失效的程序。这听起来是混乱和危险的，但事实上你没有必要花费额外的步骤去跑 cache。一个使 cache 失效的程序在 cache 关闭的情况下运行要慢 4 到 10 倍。

当你的 CPU 去设 IsC 位时，本质上必须关闭所有的中断，因为这时是不能访问内存的。

4.9.4 测试和探察

在测试、调试或 profiling 时，画一个 cache 条目的示意图是很有帮助的。 你不能直接的读 tag 的值，但对于合法的 line 有详尽的方法得到：
a. Isolate the cache.
b. 通过每条 line 的起始地址从每条 line 里取得（低位地址匹配，高位地址包括你系统的内存的物理地址范围)。每一次读取都要参考状态寄存器的 CM 位，只有该位为零时，取得的 tag 值才是正确的。

这需要很多个计算机的周期，不过对于在 20MHz 的处理器，1K 的 D-cache 对应 4MB 的物理内存，做整个查询就只需要几秒钟。

4.10 Programming R4000-Style Caches
R4000 修改了早期 cpu cache 不合适的地方。但 R4000 成功的地方就在于 cache 有多种工作模式（write-back , write-allocate)，以及拥有更长的 line。因为有 write-back 工作模式，当被 cpu 写时每一条 line 都需要一个状态位来标志这条 line 为 dirty（因此来表示很内存中的数据不同)。
对于这类 cache，我们需要 invalidate 和 write-back 操作：而且还必须保证任何 cpu 写到 cache 中的数据必须被回写到内存中。
对于诊断和维护的目的，tag 将更容易的被读写；R4000 增加了一对寄存器 TagLo 和 TagHi 用来在 cache tag 和系统管理软件之间中转数据。对于 R4000 没有直接方式读取 cache line 内的数据，当然你还是可以通过 cache 命中的方式来访问数据。CPU 可以通过执行 cache 指令来从 cache tag 内取数据到 32-bit 的 TagLo 和 TagHi 寄存器中，或者是将这些寄存器的内容写到 cache tag。图 4.3 显示这些寄存器的详细内容。
cache 的地址 tag 存有除了用来查询 cache index 的其他所有位；因此主 cache tag 的长度会因为最大物理地址（R4x000 是 36bit）和用来索引 cache 位数不同而不同。13bit 用来作为最初 R4000 的 8KB 大小主 cache 的索引，从此就再也没有小于这个位数。这样 tag 长度就有 23bit，并且 TagLo 是 24bit；在目前的 cpu 内 TagHi 总是零。这对于最小 cache 大小或是可以支持的最大物理地址是很重要的。对于 R4000，现在 TagHi 是多余的；把它设为零并忘了它。
所以 TagLo 寄存器内包含对应 cache line 的 tag 的所有位。TagLo (Pstate)还包含状态位。在绝大多数情况下（多处理器）这将变得非常复杂，但对所有 cache 的管理和初始化它足够表明当 Pstate 为零是一个合法的值来对应一条无效的 cache entry。
这个区域被后来的 cpu 占用，用来储存第二层 cache 的状态信息，但这成了一个惯例值为零是安全和合适的对于初始化。
最后，TagLo(p)是一个奇偶位，设一为整个 cache tag 偶校验。为全零的 TagLo 表明正确地偶校验。一些 cpu 忽视这个 bit，而不去检查它，而且也没有危害。

4.10.1 CacheERR, ERR, and ErrorEPC Register:Cache ErrorHandling
CPU 的 cache 是内存系统的至关重要的一部分，对于高效的实用或正确系统可以发现用额外的位来表明存储在这儿的数据的完整性是值得的。
内存系统的校验将首尾相接理想化地被执行下去;当数据一被产生或被传入系统校验位就会被计算，随着数据被存放，并在数据被使用前被检查。That way the check catches faults not just in the memory array but in the complex buses and gizmos that data passes through on its way to the CPU and back. 这样检查的方法将不在内存队列中抓住错误而在总线上，并通过这样的方式转送到 cpu 和返回。

因为这个原因，R4x000 地 CPU（设计应用于大型计算机）在 cache 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或者使用错误纠正码（ECC)。
奇偶校验是简单的使用一个额外的 bit 来对应内存中每一个 byte。一个奇偶错误可以告诉系统这个数据是不可靠的，并允许有些控制停止来代替 creeping 随机错误。奇偶校验的一个至关重要的任务就是系统开发的过程中提供巨大的帮助，因为它不能明确的指出由于内存数据完整性导致的问题。
但一个 byte 的废物将有百分之五十的机会有一个正确地奇偶校验，并且在 72 位的数据总线上的随机垃圾 256 次中将有一次没有被发现。一些系统可能会好一点。
错误纠错码计算起来将更加复杂，因为对于一个 64 位的数据将有 8 个 bit 的校验位。这将是十分的彻底，一个 bit 的错误将被唯一的指出并纠正，任何两个 bit 的错误也不会被忽视。在非常大的内存队列中 ECC 将从本质上排除随机错误。
因为纠错码一次可以检查整个 64 位的数据，所以使用纠错码的内存不能进行小于一个字的数据的写操作，被选中的小于一个字的数据必须被并入新的数据并重新计算纠错码。MIPS cpu 在 cache 关闭的情况下需要内存系统能进行小于一个字数据的写操作，这将使事情变得复杂。内存系统硬件必须将一个小于一个字数据的写操作转变为先读然后合并，接着重新计算，最后在写入的操作序列。
对于简单的系统一般的选择是奇偶校验位，而不是其他的。让采用校验方式变成可选择的这是很有意义的，这样在设计研发过程中将有利于诊断，而在成为产品时却不用付出相应的代价。
无论检查机制是运行在内存系统中还是在 R4x00 的 cache 中，cpu 将提供一个字节对应的奇偶校验位，或是对应 64 位的 8bit 纠错码，或者就是干脆没有保护。
当支持错误检测时，数据的检测位在 cache 填充时通常是直接通过系统接口到 cache 内存放而不被检查。只有在数据被使用时才检查，这样可以保证任何 cache 奇偶异常都被转交给引起它的指令，而不是转交给使用共同的 cache line。当作一个退化的情况，一个在非 cache 的取指错误会被标志成 cache 奇偶错误，这种情况会使你很混乱。
注意，系统接口标志进来的数据为没有合法的检查位是有可能的。在这种情况下，cpu 会为它内部的 cache 重新产生检查位。
如果一个错误发生了，cpu 会产生一个特殊的错误陷阱。这个 vector 会直达一个非 cache 的位置（如果 cache 内是错误的数据，它会很愚蠢地去执行 cache 内地代码)。如果系统采用 ECC，当写操作时硬件会产生纠错位，当错误时会检查得到。硬件并不知道怎么纠错；这将是软件的工作。
ERR 寄存器（如图 4.4）的格式如下：
a. ER/ED/ET/EE/EB: 这些位将区别是什么 cache(主 cache 或第二层 cache，指令 cache 还是数据 cache)发生了错误，或是它在系统接口之外。
b. PidX: 给出错误位置的 cache index。你可以取得这儿的内容用于 index 类型的 cache 操作；它将得到正确的 line，而不管 cache 是直接映射还是组相连的。

当错误发生了，ErrorEPC 寄存器指向发生错误的指令位置。ERR 寄存器保存着 ECC 位，你需要用它来纠正可以改正的错误，但在这儿我们将不再讲如何做－－因为这需要很大的篇幅，你将需要密切联系处理器手册。你将可以得到一些简单的针对 mips 运算规则的代码。
4.10.2 The Cache Instruction cache 指令
Cache 指令有着和 MIPS 存储指令类似的格式（拥有通常寄存器再加上十六位有符号的偏移地址），但表示数据寄存器的值会被译成选择区表示是什么 cache 指令。这儿没有标准的名字来表示这些 cache 操作；在这儿我就武断地使用来自 SDE-MIPS 算法库的名字，这些名字

依次是基于 SCI/MIPS 的一个头文件（include files)。选择区不是完全的位编码，但是几乎是；看表 4.2。
Cache 选择区可以让你做下面这些选择：
a. 那类 cache: 选择是 icache 还是 dcache，是主 cache 还是第二层 cache。因为没有多余的位存在,所以还没有提供的三层 cache 的选择。但这儿我要提醒你这是和 CPU 密切有关的，在 R4000 之后的 64 位 CPU 提供对 R4000 兼容性是非常有帮助的。
b. How cache is addressed:有两种不同的类型。如果是命中方式，你需要提供正常的程序地址（虚拟地址），其必须被转化。如果提供的地址确实在 cache 中，则该操作会对应相应的 cache line 被完成；如果不再 cache 中，那就什么也不用做。
另一种是 index 方式。地址低位用来直接选中某一个 cache line，而不管这条 line 现在有什么内容。这显现出 cache 内部组织的没有原则性。
cache 的维护通常是需要命中方式，而初始化时就需要 index 方式。
c. 回写（write back):如果对应的 line 是 dirty，就将数据回写到内存中去；如果不是，就像是一条 nop 指令。
d. invalidate(使无效)：将这条 line 标志为无效，使其的数据不能在被使用。同时做回写和无效是可能的；但这不是自动的，如果你需要你可以使一条 dirty 的 line 无效。所以一些应用程序可能会丢数据。
e. Load/store tags:这些操作是将对应 line 内的 tag 内容存到 TagLo 和 TagHi 寄存器中或从这两个寄存器读到对应 line 的 tag 内。
存储 tag 使用比较过时的方式（TagLo 和 TagHi 寄存器要预先设为零)，是 cache 初始化的一部分。
f. Fill:这是仅仅为 I-cache 设计的,这个操作通过特殊的内存地址来填充一条 cache line。对于填充 dcache 是没有必要的，因为当 cache 打开的时候读取没有命中时就会达到同样的效果。
g. Create data:这类操作是能够让用户能以很高的速度来写内存的排列，而避免任何的 cache 重新填充。除非你能保证在数据被使用或被回写到内存中之前覆盖这些所有的数据。这个特性对于初始化和诊断很有帮助（你将在后面初始化第二层 cache 的代码例子中看到，并知道如何清除第二层 cache 的数据)。

4.10.3 计算 cache 的大写和决定怎样配置
对于 R4x00 CPU（和绝大多数后面的 CPU）主 cache 大小和 line 的大小会同过 CP0 的 Config 寄存器可靠的给出。
但要得出你的 cache 是直接映射的还是组相连的却是相当的困难。但对于正在运行的 cache 这就不难测试出来了，你只要参考两个不可能同时出现在直接映射的 cache 中的地址，然后使用 index 类的操作去检查看它们是不是同时在 cache 内存在；当然如果你还没有初始化 cache，这是没有用的。幸运的是你可以只写同一个程序来初始化直接映射 cache 或是组相连 cache。

4.10.4 初始化程序
这儿将介绍一个很好的方法：
1.开辟一些内存对应任意的数据，但如果你的系统使用奇偶校验码或是纠错码，你必须保证这是正确的，并用这些数据填充 cache。(在算法库程序中我们保留至少 32K 的系统内存一直到cache初始化的时候;只要在cache关闭的时候去写这些内存,就能得到正确的奇偶码。)
还需要一个足够的空间来初始化的二层 cache；我们将采用迂回的方式来处理。

2.将 TagLo 寄存器设为零，这样能保证对应 line 有效的那一位没有被置起来并且 tag 的奇偶码是一致的。
TagLo 寄存器被 cache Store_Tag 指令使用，强制使对应的 line 无效和清除 tag 的奇偶码。
3.屏蔽中断，不然会有一些意外发生。
4.先初始化 Icache，然后是 Dcache。下面是初始化 Icache 的 C 代码。(你必须相信像 Index_Store_Tag_I()这样的函数或是宏能作底层的操作;它们或是琐碎的汇编代码子函数，能够运行在相应指令的机器上，或是对应 GUN C 用户通过宏调用一个 C 嵌入汇编。)

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

5.Dcache 的初始化相对来说要复杂一些，因为没有对应 Dcache 的 Index_Fill_D 操作；我们只能通过从 cache 读取数从而依靠通常的没有命中过程来达到目的。依次当 cache 填充指令对应 index 操作时，读取工程会依靠内存地址通过 tag 来命中一条 line。你必须非常小心 tag；对应 two-way 的 cache，用初始化 Icache 的循环来初始化 Dcache 会将 Dcache 的一半初始化两次，因为清除 PTagLo 会重新设置用来决定下一次没有命中时是那一组 cache Line 的位。下面是正确的方法。

```
/* 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);
```

4.10.5 Invalidating or Writing Back a Region of Memory in the Cache
对于对应一些 I/O 空间的程序或物理地址的范围，用于无效或回写的参数是不变的。
你几乎总是用命中类型的 cache 指令来使 cache 内需要的位置来使其无效或回写。如果你需要将内存一个巨大范围使其无效或回写，使用 index 类型的指令来使整个 cache 无效或回写会比较快，虽然这是一个最优化的方法，但你很可能会忽视。
我们有足够的理由这样做：

```
PI_cache_invalidate(void *buf, int nbytes)
```

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

注意这儿没有必要产生特殊的地址，只要 buf 是程序地址就足够了，但像下面的例子，如果 p 是物理地址，你就必须将其加上一个常量转化成 kseg0 范围内的地址。

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

4.11 cache 效率

从九十年代早期 cache 设计在 cpu 同一块芯片上，高速 cpu 的性能很大程度上是有他们的 cache 系统性能决定的。现在许多的系统（尤其是嵌入式系统，需要节约 cache 的大小和内存的性能），CPU 有 50－60％的时间时在等待 cache 的再次填充。就这点来说将 CPU 性能翻倍将增加应用程序 15－25％的性能。

cache 性能取决于系统在等待 cache 再次填充的总时间。你可以将之归结为两个参数产生的结果：

a. cache 没有命中的几率：是 CPU（取指令或数据存取）在 cache 中没有命中，从而需要内存中取的比例。

b. cache 没有命中需要替换造成的延时：这个延时是从内存中取数替换后到 CPU 的流水线继续下去的时间。

当然这没有必要很好的测量。举个例子，x86 的 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。当然 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 cache 的话使用效率会非常高。通过大块特殊的程序，我们就能获得 cache 没有命中的次数。

上面提到的意见将给下面指出的几种提高系统运行速度的显而易见的方法很有帮助。

a. 减少 cache 没有命中的几率：

1. 让 cache 变得更大。这是最有效的，当然也算是最昂贵的。在 1996 年，64K 的 cache 会占据高速嵌入 CPU 一半的硅面积甚至超过，所以要使 cache 的大小翻倍，你只有等待摩尔理论发明在相同的面积上做更多的门。

2. 增加 cache 的组相连。值得增加到四组，再增加下去提高就几乎看不到了。

3. 增加另外层次的 cache。当然这将使计算变得更加复杂。除了多义子系统的复杂外，第二层 cache 的非命中率会被抑制很多;主 cache 已经可以撇去 CPU 重复访问数据行为的 cream。为了使其物有所值，第二层 cache 必须比主 cache 大得多（一般来说是八倍或者更大），并且第二层 cache 的访问时间必须比内存快（两倍或者更快）。

4. 优化你的程序从而减少非命中率。如果工作处于实践阶段，这就很难说清楚了。对于小的程序优化就比较简单，但对于琐碎的程序就很费力了。但迄今为止还没有人做出一个工具可以优化任何程序。看 4.12 节。

b. 减少 cache 替换的处罚：

1.加快 CPU 获得第一个字的数据。DRAM 内存系统需要必须做许多发动工作，才能提供数据很快。使内存和 CPU 之间更加紧密，减短它们之间的路径，这样数据在它们之间来回才会更快。
注意这是唯一可以在便宜的系统中应用，并且效果不错。不过荒谬的是它很少被注意到，也许是因为它需要在 CPU 接口和内存系统设计之间考虑更多的综合因素。当 CPU 设计者在设计芯片的接口时不情愿处理这些问题，很可能因为他们的工作已经够复杂的了！
2.增加内存 burst 的宽度。这是传统上通常被应用的，是昂贵的技术，用两个或者更多的内存系统来交替存储数据；当初始化完后你就可以交替着从每一个内存系统中取数据，这样带宽好像翻倍了。第一个这样内存技术的应用是，是在 1996 年出现的同步 DRAM（SDRAM）。SDRAM 修改了 DRAM 的接口，提供更大的带宽。

c. 尽早的重新起动 CPU：这个简单的方法是在排列 cache 替换的数据时，从 CPU 没有命中的数据开始，当数据一到就马上重新启动 CPU。cache 的替换可以和 CPU 运行并行。MIPS CPU 从 R4x000 开始就应用这项技术，用一个子缓冲区来存放 cache 要替换进来的数据，并能那个字的数据是最先需要的。但只有 R4600 和其派生出来的 CPU 才体现出这项技术的好处。
激进的方式是让 CPU 绕过取数操作继续执行下去；取数的操作由总线接口控制器控制，CPU 继续运行直到它需要的数据被存放到相应的寄存器重。这被称之为没有阻碍取数，从 R10000 和 RM7000 开始被应用。
更激进的方法是，可以执行任何后续代码，只要它不依赖于还没有取来的数据，R10000 就可以不按照指令的顺序来执行。这类 CPU 应用这项技术非常彻底，不仅仅是应用到取数还应用到计算和跳转指令。

4.12 修改软件使 Influence Cache 效率更高
很多时候我们都在程序访问可知地方的基础上工作，并且我们是公平的在不强制的工作方式上操作。对于绝大多数目的我们同样可以假设，访问是恰当的随机分布式。对于工作站就必须能够支持执行足够的应用程序，这是一个公平的假设。但当一个嵌入式系统运行一个简单的应用程序，没有命中的清况好像是因为特殊的程序经过特殊的编译造成的。这是很有诱惑的，如果我们能是应用程序代码能在系统的方式下提高 cache 的效率。为了了解这是怎样处理的，你将 cache 没有命中分类，按照它们产生的原因：
a. 第一次访问：所以任何数据必须从内存中读入。
b. 替换：cache 的大小是有限的，所以当你的程序没有运行多久就会出现 cache 没有命中，需要替换掉一部分合法的数据。当程序运行时，cache 就会不停的替换掉数据然后在取。你可以通过使用大的 cache 和减小程序的大小来使替换和没有命中减到最少（其实就是程序大小和 cache 大小的比例）。
c. 从实际来讲，cache 通常没有超过四路组相连的，所以对于任何程序地址在 cache 中最多有四个位置可以存放；对应直接映射就只有一个，二路组相连的 cache 就有两个（和组相连比较 thrashing 会丢失减少速度的可能；但绝大多数研究者建议一个四路组相连的 cache 在 way 的选择上几乎不回丢失性能）。
如果你的程序会非常频繁的使用 n 段空间的数据，而这 n 段空间的地址的低位又非常接近，那么它们就会使用相同的 line。如果 n 大于 cache 的路数，那么 cache 的没有命中也会非常频繁，因为每一段空间的数据会不停的将 cache 内的其他空间的数据挤出去。

明白上面的知识，那么怎样针对程序做变化使它针对 cache 运行的更好？

a. 使程序更小：如果你能做到的话，这是个很好的主意。你可以使用适当的编译器优化（外来的优化通常是程序更大）。
b. 让程序中经常执行的那部分更小：访问密度在一个程序中不总是平均分布的。通常会有相当一部分代码几乎不被用到（错误处理，不明系统的管理），或者只被用到一次（初始化代码）。如果你能将这些很少用到的代码剥离出来，对于剩余的程序你就能得到一个很好的cache 命中率。
资格访问的方式有利于将一些频繁使用的程序区分出来，并固定放在内存的某一位子，以减少要运行时的放置。这样至少这些经常使用的程序不会因为 cache 的位置而相互碰撞。
c. 强迫一些重要的代码或数据常驻 cache:一些机器的能允许一部分 cache 保护它们所拥有的数据不被替换。这部分代码一般是中断处理或是其他自关重要的软件中确定要执行的。这些代码或数据一般被保留在二路组相连 cache 的其中一组中（这样当系统重其后，cache 就像是一个直接映射的 cache).
我很怀疑这个方法的生存能力，我也不知道有那些研究支持它的有效性。系统重起后的损失很可能大于执行那些保留代码的获利。cache 上锁很可能就像一个不确定的市场工具来限死顾客对 cache 启发式特性的渴望。这个渴望是可以理解的，随着程序越快越复杂越大，但 cache 毕竟只是影响结果的一部分因数。
d. 安排程序避免碰撞：上面提到的让程序的执行部分变的更小，这对于我来说太难维护所以不是个好的主意。而且对于组相连的 cache（尤其是两路的）使这个方式更加没有意义。
e. 让那些很少用到的数据和代码不经过 cache：这看起来很有吸引力，让 cache 只给那些重要的代码或数据服务，排除那些只用一次或很少使用的代码和数据。
但这几乎总是一个错误。如果数据真的很少使用，那么它们不可能一开始就在 cache 中。因为 cache 取数时总是以一条 line 的长度 4 或 16 字为单位,所以即使是传送只使用一次的数据也能有很高的速度；burst 替换比一个字的访问几乎不多花时间，并且能给你免费提供另外的 3 或 5 个字的数据。

简而言之，我们将介绍下面的内容作为一个起点（除非你已经有很多实践和很深的想法，你才可以放弃)。开始我们先认为除了 I/O 寄存器 cache 都是打开的，并且很少使用远程内存。在你试着做预测前，先搞明白 cache 对你的应用程序有什么启发。第二在硬件上排除任何问题。也没有任何软件辅助收回因为高的 cache 替换率和小的内存带宽而损失的性能。尝试从新组织软件而降低 cache 的非命中率，而不能增加其长度和使其变复杂，但也要明白一开始收获是很小并且来之不易。也试着在硬件上做优化。

4.13 Write Buffers and When You Need to Worry
通常使用 write－though cache 的 32 位 MIPS CPU，其每一个写操作都会立刻直接写到主存中，如果 CPU 要等待每一个写操作完成才能继续，这将是是一很大的性能瓶颈。
C 语言程序编译给 MIPS 使用，平均有 10%的指令是存储指令；但这些操作可能可以趋向于合并为 burst，举个例子在一个函数开始时的现场保护（保存一些寄存器)。DRAM 内存通常有这样的特征，一组中第一个写的通常会花费很长时间（这些 CPU 一般是 5 至 10 个时钟周期），而第二个和后边的就会相应很快。
如果 CPU 简单地等待每一个写操作完成，这对性能地打击很大。所以通常会提供一个回写地缓冲区，先入先出地存入要写地数据和地址。
使用 write-through cache 的 32 位 MIPS CPU 很倚重回写缓冲区。在这些 CPU 中，在 CPU 的时钟频率达到 40MHz 时能缓冲四次的队列将很难提供很好的缓冲。
后来的 CPU(有 write-back cache)的缓冲区能直接保存需要回写的 line，并且提高非 cache

写的时间。
许多回写缓冲区操作的时间对于软件来说是透明的。但有时候编程人员要注意下面的情况：
a. I/O 寄存器访问的时间：这对所有 MIPS CPU 都有影响。当你执行一个向 I/O 寄存器的写操作，这会有一个不能确定的延时。和 I/O 系统之间的其他通讯可能会很快，举个例子在你告诉设备不要再产生中断后，你还是可能会看到一个活跃的中断。在其他例子中，如果 I/O 寄存器在一个写操作后需要一定时间来恢复，那么在你开始计算这个延时前，你必须保证回写缓冲区是空的。这儿你必须保证 CPU 等待直到回写缓冲区腾空。定义子程序来做这项工作是个好习惯；子程序叫 wbflush()，这是个传统。看后面的 4.13.1 节，如何实现。

上面描述了在任何 MIPS R4x000（MIPSIII ISA）上可能发生的。还针对整个 IDT R3051 家族，和绝大多数流行的嵌入式 CPU。但在一些早期的 32 位系统中，更怪的事可能发生：
a. 读操作赶上写操作：当一条取数指令（非 cache 或是没有命中 cache）执行时，回写缓冲区不是空的。CPU 需要选择：是等写操作完成还是将内存接口给取操作使用？让取操作先做将提高效率，因为 CPU 需要等待直到取的数据到来。这是一个好的机会，写操作被压倒，但它后面还是可以和 CPU 并行。
最初的 R3000 硬件将这个选择留给系统硬件来决定。从 IDT 开始的绝大多数 MIPSI CPU 不允许读操作压倒写操作，写操作有没有条件的优先权。绝大多数 MIPSIII CPU 不允许读操作被压倒，但软件也没有必要对此进行过多的考虑。看 8.4.9 节关于 sync 指令的描述。
如果你确认你的 MIPS I CPU 没有无条件的写的优先权，那么当你在处理 I/O 寄存器时，必要的地址检测也不能帮到你；因为早期的一个向不同地址的写操作还没有完成，那么这时候的取操作就会产生错误。在这中情况下你就需要调用 wbflush()。
b. 字节合并：当缓冲区注意到一些部分字的写操作是写向相同的字地址，缓冲区会将这些写操作合并成一个简单的写操作。这并没有被所有的 R3051 家族的 CPU 所采用，因为它可能在对 I/O 寄存器的写操作时产生错误。
如果将你的 I/O 寄存器映射成每个寄存器对应一个独立的字地址，这就不是一个坏注意了。但你不可能总这样做。

4.13.1 执行 wbfush
除非你的 CPU 是上面提到的特殊类型的一种，你能保证在执行针对任何地址的取操作时，回写缓冲区是空的（这会暂停 CPU 直到写操作完成，取操作也完成）。但这样效率不高；你可以通过使用最快的内存来最小限度来克服一些。
对于那些从来都不想考虑这些的人，一个写操作后面紧跟着一个针对相同地址的取操作（如果你是运行在 MIPS III 或其后面的 CPU，需要在两个指令间加入 sync 指令），需要先清空回写缓冲区（很难看到如果 CPU 没有这个动作也能执行正确）。
一些系统通过硬件信号来指明 FIFO 是不是空的，连到输入接口让 CPU 能立刻得知。但不是到目前为止的 CPU 都这样做。

CAUTION! 一些系统通常在 CPU 外面也有写操作的缓冲区。任何总线或内存接口自夸的写加速也会出现相同的特征。写缓冲区在 CPU 外和在 CPU 内一样，也会给你带来相同的问题。

4.14 其他的关于 MIPS CPU
虽然你可能永远也不用知道这些，但我们还是有许多理由要谈到这些。

4.14.1 多处理器的 cache 特征
这个书的讨论将仅仅针对单 CPU 的系统。感兴趣的可以读相应的文档(Sweazey 和 Smith 著，1986)。

4.14.2 Cache Aliases
这个问题只会影响这样一类的 cache,用于产生 index 的地址和存放在 tag 内的地址不一样。在 R4000 类型 CPU 的主 cache 中，index 是由虚拟地址长生而 tag 内存放的是物理地址。这对性能很有好处，因为 cache 查询和地址转化可以并行，但这也可能导致 aliases。
绝大多数这些 CPU 可以按照 4KB 一页的大小来转化地址，而 cache 大小是 8KB 或者更大。两个不同的虚拟地址可以映射成对应一个物理地址，而且这是两个连续页，我们可以假设它们开始地址分别是 0KB 和 4KB。如果程序访问地址 0KB,那么数据会被读入到 index 为 0 的 cache 中。我们在假设要地址 4KB 来访问相同的数据，cache 就又会从内存中取数并保存到 cache 中，但这次 index 却为 4KB。这时在 cache 中针对相同数据有两个备份，一个被改变了，另一个却不会受到影响。这就是 cache 的 alias。
MIPS 第二层 cache 是物理地址来产生 index 和被存放到 tag 中，所以它们不会产生上面的问题。
不过，避免这个问题比改正它更容易。如果任意两个不同的虚拟地址能产生相同的 index，那么这个问题就不会出现了。对于 4KB 一页，只要保证用来产生 index 的最低 12 位地址一致；只要保证不同虚拟地址对应的物理地址页大小等于主 cache 大小的模。如果你能保证虚拟地址是 64KB 的倍数，这就不可能产生这个问题，你也不会遇到麻烦。

第五章 异常,中断,初始化

在 MIPS 体系结构中,中断,陷入,系统调用,以及其他中断程序正常执行的事情统统被称为异常.异常在 MIPS 体系结构中被同一种机制处理.异常包括:

外部事件.包括中断,读总线错.在有外部事件时,中断被用来引起 CPU 的注意.使用中断比使用 CPU 轮询机制来得快且更有效.
中断是唯一由 CPU 执行以外的事件引起的异常.因为我们不能通过注意来避免中断,在必要是,我们只能用一种软件的办法来禁止中断.

内存翻译异常.当没有合适的物理地址对应虚拟地址时,或当写一个有写保护的页时,会发生此种异常.操作系统会监查内存翻译异常的具体原因.某些异常是由于应用程序访问了非法内存.操作系统会终止应用程序的执行以保护其他应用程序.良性的内存翻译异常可以触发操作系统执行从复杂到简单的一系列操作:操作可以复杂到装入一个需要时调入内存的虚拟页,或简单到扩大栈的空间.

其他需要内核更正的不寻常情况.一个例子是浮点指令:当硬件无法处理某些困难和少见的操作符和操作数的组合时,硬件会产生一个异常,寻求软件模拟.这类情况比较模糊,因为不同的对这类情况会有不同的处理意见.未对齐的装入在某些操作系统中被由软件处理,在另外一些操作系统中被当做错误.

程序或硬件检查出的错误.包括非法指令,在不正确的用户权限下执行的指令,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,用户模式下访问超出用户段(KUSEG)地址.

数据完整性错误.很多 MIPS CPU 不断对来自总线和缓存的数据作字节校验或字校验.校验错在 R4000 及以后的 CPU 上产生一个特殊异常.

系统调用和陷入.某些指令只是用来产生异常.它们提供了一种进入操作系统的安全机制(系统调用,条件陷入,断点).

某些事件不产生中断,尽管大家认为它们会.比如写总线错.CPU 把数据和地址放入写缓冲中,然后继续执行.写操作可能在几个时钟周期后发生.在这种情况下,很难判断到底是哪条指令产生写错误.某些系统利用外部机制来解决这个问题.这种外部机制有可能会产生异常来引起 CPU 注意.

更为有意思的是,在大多数32位的CPU上,缓存中的校验错不产生异常.错误被放在一个状态寄存器位 SR(PE)里,你必需自己去察看它.在 R3000(32 位的 CPU)里,缓存校验在以后加入用于调试目的.

在这一章里,我们将要讨论:CPU 如何决定产生异常,软件如何正确处理异常,为什么 MIPS 的异常叫做精确异常,异常入口点,以及一些软件编程约定.

在嵌入式系统中,来自 CPU 外部的硬件中断最常见的异常异常,要求得到及时处理,并很容易导致不易觉查的错误.异常嵌套---在一个异常的处理中发生另一个异常---可能引起一些特殊的错误.

系统重置(RESET)后,MIPS CPU 靠异常启动,所以 CPU 启动也在这一章中讨论.在这一章末尾,我们会讨论一些相关的话题,如软件模拟机器指令,构造一个信号量用以提供任务到任务的通信.第十二章还包含一个有关异常处理的有注释的程序.

第一节 精确异常(PRECISE EXCEPTION)

我们会在 MIPS 文档中看见精确异常这个词.精确异常非常有用.为了讲清楚为什么,我们应该看看它的对立面:

在一个以优化性能为主要目的的流水线中(或者是用于指令并行执行的设计中).系统的顺序执行只是一种抽象.如果硬件不是设计得特别聪明,中断讲使我们看到程序不是顺序执行的.

当一个异常发生,系统的顺序执行被中断时,流水线(PIPELINE)的 CPU 将会有几条指令出于流水线的不同阶段(STAGE).因为我们不想中断处理破坏程序的正常执行,对于没有执行完的指令,我们必需记住它们执行到哪一个阶段,以便在中断处理之后能恢复程序执行.

如果 CPU 是精确异常的,那么异常的软件处理就会非常简单.对于一个精确异常的 CPU,在异常发生时,我们都会有一个引起异常的指令(EXCEPTION VICTIM).该指令前面的所有指令都

以执行完,该指令以及该指令以后的指令都不会有任何软件值得考虑的负作用.所以软件作异常处理时,可以完全忽略指令的非顺序执行(即假定系统时顺序执行的---译者)

对于 MIPS 体系结构,几乎所有的异常都是精确异常,陈述如下:

能准确定位恢复程序执行的正确位置.在任何异常产生后,CPU 的控制寄存器 EPC 指向中断处理后恢复程序执行的正确位置.在绝大多数情况下,它是产生异常的指令,如果产生异常的指令是转移指令的延迟单元(BRANCH DELAY SLOT),EPC 则指向该延迟单元的前一条转移指令.异常处理后,讲返回该转移指令.如果 EPC 错误的指向延迟单元,则异常处理后讲会从延迟单元开始执行,从而导致程序的错误执行.如果产生异常的指令是转移指令的延迟单元,原因寄存器位 CAUSE(BD)将会被设置,因为有些异常处理程序需要知道是那条指令引起的异常,在这种情况下,它是 EPC+4.

看起来找出产生异常的指令可能很容易,但在有些级数很多的流水线 CPU 上这件事并不容易.

异常发生的顺序与指令的顺序相同.在非流水线的CPU上,这是很显然的.在流水线的CPU上,异常可能会发生在指令执行的不同阶段.产生潜在的问题.比如,如果一个读内存指令产生一个地址异常,这个异常一直要到读写内存(MEM)阶段才产生.如果它的后一条指令在取指令(IF)阶段就产生错误,则后一条指令讲想产生异常.从而破坏异常发生的顺序很指令的顺序相同这个约定.

为了避免这个问题.被发现的异常情况一直要到确认有异常情况的指令的前面的所有指令都不产生异常时才产生异常.在发现异常情况时,该情况只是被记下来沿着流水线传递下去直到某一级.如果在这个过程中以前的指令产生的异常被发现,该异常情况仅仅被简单的忽略掉.这样,以上的问题就被解决掉了.---该情况很有可能在中断处理返回后再一次发生.

产生异常的指令的后续指令无效.因为MIPS是流水线CPU,在发现异常时,产生异常的指令的后续指令也被执行了.但是我们可以确认这些产生异常指令之后的指令不产生系统程序员能看得到的后果.处理完异常情况后,程序会从 EPC 继续执行,就象从来没产生过异常一样.

在一些情况下,MIPS 的异常不是精确异常.比如,整数相乘并不响应异常(注解).这个问题可以通过安排指令顺序被解决.指令顺序在汇编语言中被强制规定.

MIPS 的精确中断代价很大.应为它限制了可以流水线执行的可能性.这种限制对于浮点运算尤为严重,因为浮点运算需要很多级才能完成.只有系统确认不会产生异常是才会让浮点指令进入算术逻辑单元(ALU).

第三节 中断向量，软件中断处理开始的地方

绝大多数的 CISC 处理器有专门的硬件或不为人知的微指令分析异常，根据异常产生的不同原因让 CPU
进入不同的入口地址。MIPS 处理器在这方面作的很少。如果你认为这是一个严重不足的话，请看下面分析。

首先，具有中断向量表的中断处理机制不如我们所希望的那样有用。在绝大多数操作系统中，中断处理程序共享代码（比如在中断处理入口存储寄存器的值等等）。在 CISC 的 CPU 中我们常看到 CISC 硬件或微指令把 CPU 分派到不同的入口地址，操作系统记住中断原因后再跳转到同一地址。

其次，如果不用微指令，很难想象硬件能做多少复杂的工作。在 RISC 机器上，指令速度足够快，完全可以作为我们的首选。

再这里和其他地方，我们应该记住 RISC 处理器比外部设备快得多。通常中断处理程序会读外部寄存器的值，而对于 9 0 年代的 CPU 而言，外部总线的时钟周期（即读外部寄存器的值所花的时间－译者）大致是处理器时钟周期的 2 0 到 5 0 倍。所以很容易写一端由外部寄存器的值决定处理器执行什么中断处理程序的代码，在这段代码中，读外部寄存器的值花主要时间。因此，软件处理不会成为性能的瓶颈。

即使是 MIPS 处理器，也不是所有的中断都是同等对待的。随着处理器体系结构的进步，差别将会拉大。列举如下：

用户地址 TLB（ translate look-aside buffer）填充。在被保护的操作系统中，有一个经常发生的异常，它和地址翻译有关（见第六章）。TLB 硬件只保存有有限的虚拟地址和物理地址的对照表，因此在使用虚拟地址的操作系统中，很复杂的程序所用的虚拟地址有可能不在 TLB 中。我们把这个事件称为 TLB miss（因为 TLB 是一种有软件管理的缓冲）。

使用软件来管理这种情况在精简指令处理器刚被推出时是有争议的。MIPS 处理器为软件管理 TLB miss 提供了有力的支持。在这种异常处理中，硬件提供了足够的支持所以软件只需要 1 3 个时钟周期就足够了。

这种经常有用到的程序被提供了特殊的入口地址，所以它能被优化的很好，不必判断到底是哪种异常发生了（Amdal's law）。

64 位地址的 TLB 填充。在 64 位的处理器上，对于想充分利用 64 位地址空间的程序，它们的地址翻译会用到稍微有些不同的寄存器安排和 TLB 填充程序。MIPS 把它们称 XTLB。这个设计同样是为了使程序运行更有效。

不在缓存中的中断处理程序入口点。为了让异常处理更有效，中断处理程序的入口点应该在缓存中。但在系统启动的时候，这几乎是不可能的。如果你想有一个健壮的自较验的启动序列你必需在启动时使用未被缓存的，只读的中断入口点。在 MIPS 处理器中没有不缓存模式，只有不能被缓存的内存区域，所以我们应该使用 SR(BEV)位是中断入口点被冲定位到未被缓存的，启动安全的 kseg1 区域。

数据校验错。R4000 和以后的处理器都检测数据错误并在检测到数据错误时产生一个陷入。数据错通常在数据从内存中读出的时候发生，在数据从缓存中被读出的时候被检测到。在数据出错时，用被缓存的入口地址显然是不明智的。所以此时无论 SR(BEV)位被设置与否，都应该用为被缓存的入口地址。

系统重置。从很多方面来说，把系统重置看作一种异常都是有道理的。特别是在考虑到 R4000 及以后的处理器用同一个中断入口地址处理冷启动和热启动的时候。事实上，不可屏蔽中断应该被看作是一个较弱的热启动，和热启动相比，它的唯一区别就是，必须在当前指令被完成之后才生效。

所有异常的入口地址都在 MIPS 内存映象的未被翻译的区域。如果入口地址是未被缓存的，他就位于 kseg1 中。如果入口地址是被缓存的，他就位于 kseg2 中。异常处理程序的入口地址见表 5.1。如果处理器使用 64 为地址，则对相应的 32 位地址作符号扩展：如果 32 位地址是 0x8000,0000,则 64 位地址是 0xffff,ffff,8000,0000.表 5.1 描述的是 32 位地址的入口地址。

中断入口地址之间差 128 个字节，因为 MIPS 的设计者认为对于基本的中断处理，32 条指令应该足够了。这样，我们就省掉了跳转指令，而有不浪费太多的内存。

第六章 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，为了说清楚内存管理硬件，我们得从 MIPS R2000 所寻求实现的 unix 风格的虚拟存储系统开始讲起。本章的后面我们还会讨论一下相同的硬件如何在其他环境下工作。

早期的 MIPS CPU 定位于支持运行在 UNIX 工作站与服务器上的应用程序，因此内存管理硬件被构想为一个最小化的能帮助 BSD UNIX——一个经过完善设计并拥有充分多虚拟存储需求的操作系统的典型——提供内存管理功能的硬件。很明显的是，这些设计者们十分熟悉 DEC VAX 小型机，并且在从这种体系结构中获取了众多思路的同时，也摒弃了许多复杂设计。尤其是许多 VAX 使用微代码来解决的问题，在 MIPS 中被交由软件处理。

本章中我们将从 MIPS 的设计起点开始，面对着一个 unix 类型的操作系统以及它的虚存系统的众多需求。我们将会展示一下 MIPS 的硬件是如何满足这些需求的。结尾时，我们会讨论一下在不能像通常一样使用内存管理硬件的嵌入式系统中，您可以采取的几种使用方式。

内存地址转译硬件（下面我们将称其 MMU，全称为 memory management unit）有几类不同用途：

重定位（Relocation）：程序的函数方法和预先声明的数据地址均在编译期间决定，MMU 允许程序在任何物理地址运行。

为程序分配内存：MMU 可以从物理内存里许多零散的页中创建连续的程序空间，使我们能从一个充满固定大小页面的池里分配内存。如果我们不停分配释放大小不一的内存块，就会碰上内存碎片问题：我们不得不止步于一个布满“小孤岛”的内存空间，无法满足对较大块内存的申请要求，哪怕此时所有的空闲空间之和是足够的。

隐藏和保护：用户级程序只能访问 kuseg 内存区域（较低的程序地址）内的数据。这类程序只能在操作系统所许可的内存区域中取得数据。

此外，每一页可以独立的指定为可写权限或者写保护权限；操作系统甚至可以停止一个意外的写覆盖代码空间的应用程序。

扩展地址空间： 有些 CPU 不能直接访问它们拥有的全部物理空间。尽管 MIPS I 系列 CPU 是真正的 32 位体系结构，它们却布局了地址映射，使得未被映射的地址空间窗口 kseg0 和 kseg1（它们不依赖 MMU 进行地址转换）落在了物理内存的开头的 512M 内。如果你

要访问更高地址，则必须通过 MMU。

内存映射对程序的适应化：在 MMU 的帮助下，你的程序能够去使用适合它的地址。同一段程序的许多份拷贝可能会同时运行在一个庞大的操作系统里，令它们去使用相同的程序地址变得更容易。

调页能力：程序可以好像已经得到它们所申请分配的所有资源一样正常的运行，而操作系统实际上只分配给它们当前所需的资源。访问未分配空间的程序会导致一个交由操作系统处理的异常（exception），操作系统此时才在这块内存中装入适当数据并令应用程序继续运行。

UNIX 内存管理工作的本质是为了能运行众多不同的任务（即 multitasking——多进程），并且每个任务各自拥有自己的内存空间。如果这个工作圆满完成，那么各任务的命运将彼此独立开来（操作系统自身也因此得以保护）：一个任务自身崩溃或者错误的做某些事不会影响整个系统。显然，对一个使用分布终端来运行学生们程序的大学而言，这是一个很有用的特性；然而不仅如此，甚至是要求最严格的商业系统环境也需要能够在运行的同时支持实验软件或原型软件一并进行调试和测试。

MMU 并不仅仅为了建立巨大而完备的虚拟存储系统，小的嵌入式程序也能在重定位和更有效的内存分配里受益。如果能把应用程序观念上的地址映射到任何可获得的物理地址，系统在不同时刻运行不同程序就会更加容易。

多进程和隔离不同进程地址空间一直都在向更小的计算机上移植，目前在个人电脑以及英特网服务器端都已经十分普通。

嵌入式应用中常常会明确的运用多进程机制，但几乎没有多少嵌入式操作系统使用隔离的地址空间。或许这归咎于这种机制在嵌入式 CPU 以及它们上面的操作系统上用处不大并且带来不稳定性，因而显得不那么重要。

MIPS 这种如此之必要以致于导致在 1986 年时工作站 CPU 变的廉价起来的简单机制，或许也可以被证实跟 90 年代后期嵌入式系统的兴起有一定关系。甚至是很小的应用，也被迅速增长的代码大小所困扰，需要使用所有已知的手段来控制软件的复杂度；这种由 MIPS 首创的灵活的基于软件的方法看来能提供任何所需空间。仅仅几年前，CPU 的厂商们在定位嵌入式市场时还很难确定 MMU 是否值得包括进去；然而到 1997 年，微软推出的无法在没有内存管理硬件的环境下运行的 Windows/CE，已被视为针对嵌入式所面对的各种困难的一个成功解决方案。

6.1 大型计算机上的内存管理

或许从一个类似 unix 系统的内存管理系统的整个工作开始讨论是最容易的(选择 unix 作为研究是因为：尽管它体积庞大，却比 PC 上的操作系统简单的多)。在图 6-1 中展示了其典型特征。

6.1.1 基本的进程空间布局和保护

图 6-1 中最宽的分隔线是在低半部分——标明“用户程序可访问”的那部分——以及剩余部分之间的。程序的空间中的用户可访问部分就是我们在 2.8 节所描述的通常在 MIPS 内存映射中称为“kuseg”的部分。所有的高位地址内存都保留给操作系统。从操作系统的角度来看，地址低半部分是一个用户程序可以随心所欲使用的安全的“沙盒”（sandbox）。假如程序运行错误并且毁坏了自己所有的数据，其他程序并不用担心受到影响。

从应用程序的角度来看，这块区域可以随意用来创建独享的复杂数据结构来继续自己的工作。

在用户区域内部，也就是在“沙盒”的内部，操作系统给有需求的程序提供更多的栈空间（由于栈在暗中向下增长）。同时也提供一个系统调用，用来从一个以“预先声明数据区域”（declared data）最高地址为起始地址并且不断增长的地址空间——人们称之为“堆”（heap）——当中来获取更多数据空间。“堆”用来实现诸如 malloc()这样的用于给应用程序提供大块额外内存的库函数。

用来构建堆和栈的内存块应该小到足以使系统节约内存，但同时也必须大到可以避免过多的系统调用或者访存异常的产生。不过，在每一次系统调用或访存异常时，操作系统会有机会监督应用程序的内存消耗。操作系统可以增强限制以确保应用程序不会获取过多的内存以致于威胁到系统中关键的运行活动。

在 unix 类型的系统中，进程在操作系统内核中拥有自己的识别符；为了确保应用程序只能做它们被允许的事情，绝大多数内核服务以特殊子函数的形式（即系统调用）提供出来，应用程序调用时也必须遵从一定的特殊规定。

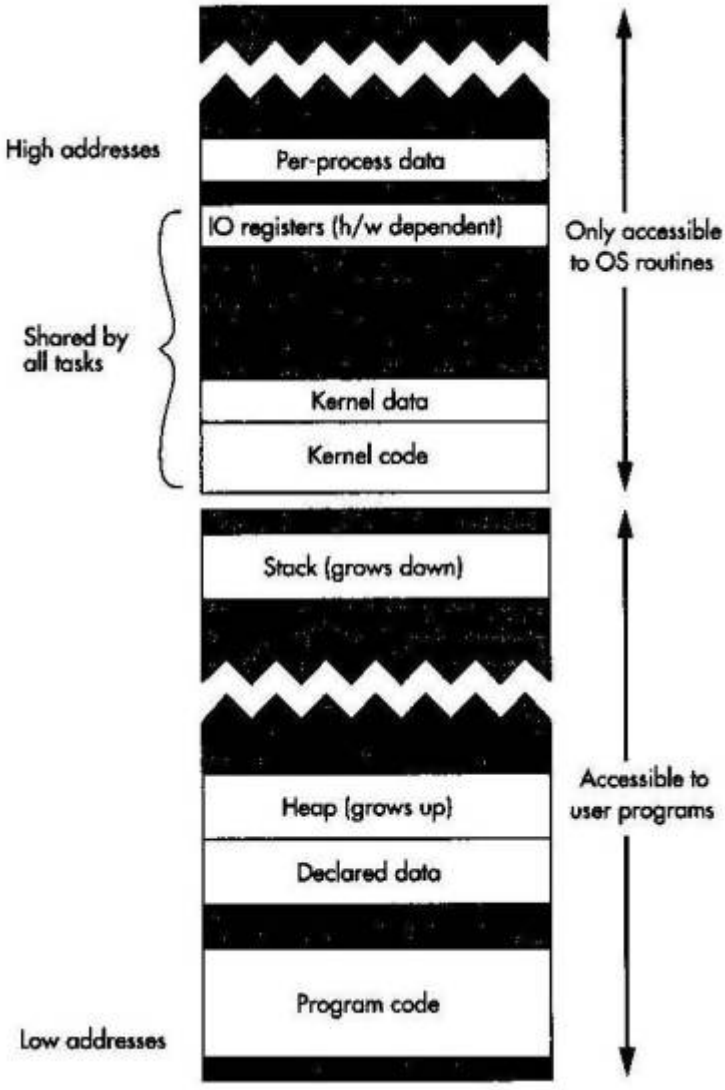


图 6.1

操作系统自己的代码和数据显然不能被用户空间的程序访问到。在某些系统里，这是通过把它们完全安置于一个隔离的地址空间内来完成的；在 MIPS 上运行的操作系统则与用户程序共享同一地址空间，当 CPU 运行在用户级权限下时，访问内核的空间是非法的，将会导致一个异常产生。

需要注意的是，尽管每个进程的用户空间会映射到专属于本进程的物理存储空间上去，操作系统的空间却往往是共享的。大部分的操作系统代码以及资源在所有进程看来位于相同的地址——操作系统内核内部是一个多线程单地址空间的——而每个进程的用户地址空间则位于专属自己的隔离空间。应用程序发出的系统调用在内核里的运行过程是完全可信赖的，而应用程序则根本无须被信任。

用户空间的有效部分是被分开的，栈位于空间顶端，而代码和静态编译的数据位于底部。这就使得栈可以向下增长（这是隐式的，由于程序的运行中函数参数的累积）而数据空间能够向上增长（这是显式的，由于程序调用了分配内存的库函数）。操作系统能够为栈或数据空间分配更多的内存并映射到合适的地址上去。

请注意，为了使程序能够使用庞大的数据空间，通常会让栈从用户空间所允许的最高地址开始向下增长。地址转换方案中必须要妥善应对这种在大跨度的范围内使用地址空间（在被使用空间中有一个巨大空洞）的地址映射特征。

实时系统中为了寻求效率和更多的共享函数，机制更加复杂化。大多数系统把应用程序的代码映射为“只读”（read-only），这意味着这些代码可以被许多进程安全的进行共享——许多进程运行同一应用程序的情况也很常见。

许多系统不仅仅共享整个应用程序，还可以共享通过库调用来访问的程序段（共享库）。目前我们还是先暂不讨论这所引发的另外一大堆问题吧。

6.1.2 把进程空间映射到物理内存

支持这个模型需要什么机制呢？

MIPS 体系结构或多或少地要求程序（不管是应用程序还是内核方法）的地址空间在编译连接期间固定下来。这意味着应用程序在构建时不可能明确使用不同的地址——在我们希望运行同一应用程序的不同拷贝时也是如此。因此，当程序运行时，它的地址会映射到一个在程序装入时就已经由操作系统所固定下来的物理地址上。

尽管在进程上下文切换时更新所有的地址映射信息是可行的，但其效率相当之低。替代办法是：我们给每个进程一个编号（在 unix 里称作“进程 ID”，但更准确的叫法应该是“地址空间 ID”或者简称为“ASID”）。每个进程里的任何地址都在暗中被进程的 ASID 扩展后产生一个唯一的待转换地址。ASID 需要在进程被新调度执行时装入 CPU 的一个寄存器，使得硬件可以来使用它。

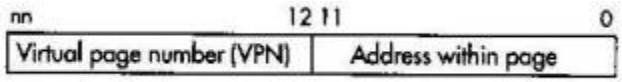
映射机制还令操作系统能够把用户空间的不同部分区分对待：应用程序的某些空间（一般是代码部分）被映射为只读，而其它某些部分则可以暂不映射并且对其访问能引起“陷入”（trapped），这意味着一个胡乱运行的程序可以更早被停住。

进程地址空间中的内核部分通常是被所有进程共享的，而且这一部分绝大多数内容映射为常驻的操作系统代码和数据。由于这些代码连接为在这些地址运行，不需要灵活的映射机制，大部分 MIPS 上运行的内核会把它们的绝大部分代码置于这块在这一体系结构里具有固定映射的地址空间中。

6.1.3 页映射最佳

为了映射地址人们尝试过很多特殊办法，通常使用“基地址/范围”二元组来保证地址的正确性。然而如果以提供给程序恰好其所需大小的内存的方式来进行内存映射，尽管这很明显为应用程序提供了最优服务，却会迅速导致可用内存变为零散的具有难以使用的大小的内存碎块。所有的实际系统都对内存以页（page，一种固定大小的内存块）进行映射。页通常是

2 的幂次大小，4K 大小得到了压倒性优势的使用频率。
在 4K 的情况下，一个 CPU 地址可以简单的映射为这样：



“页内地址”（图中 Address within page 部分）的几位不需要转译，因此内存管理器件只需要去处理地址高位的转译，即把通常称作“虚页号”（图中 Virtual page number，简称 VPN）的部分转译为实际物理地址的高位（即 physical frame number，或简称 PFN，没人能想得起来为啥不叫 PPN）。

6.1.4 我们真正想要的

映射机制必须使一个程序能断言某个地址在其自己的进程空间或地址空间内，并且能够高效的将其转换为真实的物理地址以访问内存。
一个好主意是使用一个含有整个空间内所有页的入口（entry）的表（即页表），每个入口包含这个页的正确物理地址。这很明显是个相当大的数据结构，因而不得不存放于主存之中。不过这带来两个严重问题。
第一，每次取出或存入数据我们都需要访问两次内存，就性能而言这显然是没什么好指望的。大概您已预见到这样一个答案：我们可以使用一个快存（cache）来存储这些入口，仅仅在我们未命中快存时才去访问常驻内存的表。由于每个快存的入口覆盖了 4KB 的内存空间，我们似乎可以就这样得到一块未命中率出奇低而自身又相当小的快存。（现在要介绍一下，快存很稀少而且有时被称作“查找缓存”（lookaside buffers），因此内存转译快存就被称作“转译查找缓存”（translation lookaside buffers）或简称为 TLB；这个缩写更加常用）。
第二个问题是页表的大小；对一个划分为 4KB 大小页面的 32 位应用程序的空间，将产生 100 万个入口，这会占去将近 4MB 的内存。我们有必要找些办法让这个表小一点，否则就剩不下多少内存给程序使用了。
我们将在这样一个前提下讨论不同的解决方案：真实运行中的程序会在地址空间中留有巨大的地址空洞，如果我们能有一种方法，避免用物理内存来存储表中的这些空洞，情况看上去会好的多。
现在我们有解决办法了，本质上，这来自于 DEC 在 VAX 小型机上所使用的内存转译系统，它给绝大多数的并发体系结构带来了深远的影响。图 6-2 中对其做了概括。

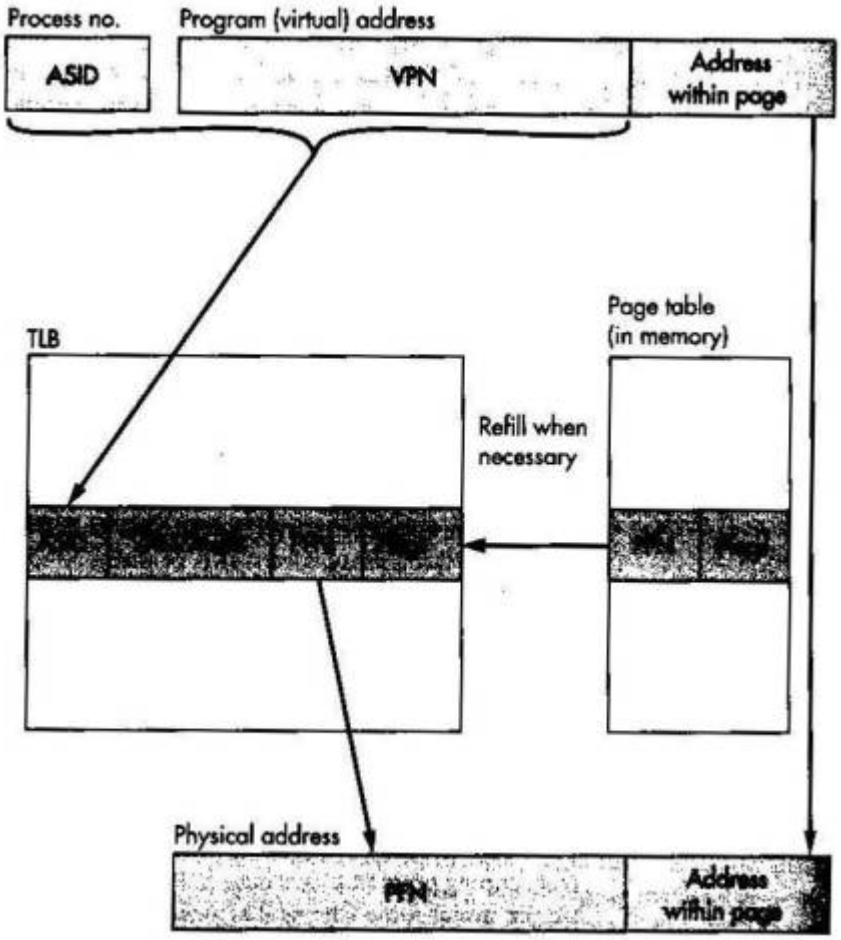


图 6.2

硬件的处理过程大致上是这样的：

一个虚地址被分割为两部分，低半部分地址位（通常为 12 位）不经转译直接通过，因此转译结果总是落在一个页内（通常 4KB 大小）。

高半部分地址位，也就是 VPN，会在前面拼接上当前运行进程的 ASID 以形成一个独一无二的页首地址。

我们在 TLB 中查找是否有一个本页的转译项在里面。如果有，那么我们将得到对应的物理地址的高位，最终得到可用地址。TLB 是一个做特殊用途的存储器件，可以运用各种有效的方法来匹配地址。它可以参考一个全局的标志位来在查找某些入口时忽略 ASID 位，因此这些 TLB 入口可以用来映射所有进程中的某一段共享的虚地址空间。

类似的，VPN 也可以在存储的时候使用某些掩码位，使 VPN 中某些位在匹配过程中被排除在外，这使得 TLB 入口能够映射更大范围的虚地址。

在某些 MIPS MMU 中这两种特殊机制均被采纳。

通常在 PFN 中还存储了一些额外的位信息（flags）以用于控制哪些访问可以被允许——最明显的，允许读操作而不允许写操作。我们会在 6.2 节中讨论 MIPS 体系结构的标志位。

如果在 TLB 中入口匹配失败，那么系统必须定位或者分配一个适当的入口（使用常驻内存页表的相应信息）并将其装入 TLB，然后再次进行一次转译过程

在 VAX 小型机中，这个过程是被微代码（microcode）所控制的，对程序员而言整个过程完全是自动进行的。

6.1.5 MIPS 如此设计的起源

为了在尽可能少使用硬件的前提下提供一套与 VAX 相同的功能，MIPS 的设计者们需要找些好办法。由微代码控制的 TLB 重装入（refill）是不能接受的，因此他们勇敢的迈出了一步：把这个工作交给软件来完成。

这意味着除了有一个寄存器用来存放当前的 ASID，MMU 器件仅仅是个 TLB 而已，也就是一个简单的高速、定长的转译表。系统软件可以（通常也就是如此）把 TLB 作为一个快存来面向常驻内存的页表，然而 TLB 硬件本身并不能把自己当作快存来使用，而只能这样：当某一个地址无法进行转译时，TLB 会触发一个特殊的异常（TLB 重装入异常）来引发调用软件程序。不过，TLB 的细节设计和相应的控制寄存器上都作了十分周密的考虑，以帮助软件更加富有效率的运行。

6.2 MIPS TLB 的特点

MIPS TLB 通常都是在芯片上实现的：即使在快存命中的情形下，内存转译也这一步仍然必须进行，因此在机器上这是一个十分重要的“关键路径”（critical path）。这意味着它必须很小，尤其在早期那个年代，因此，把它的规模控制的很小就十分明智。

基本上这是块全相连的存储单元。每个入口都是一块拥有键值（key）域和数据域的关联存储器；当你提供某个键值后，由硬件来进行匹配并给出匹配成功的入口内的数据。通常全相连存储器效率很高但在硬件上过于浪费，而 MIPS 系列的 TLB 含有 32 到 64 个入口不等；这种规模的存储量在芯片设计中是相对容易处理的。

R4000 风格的 CPU 至今都在使用这样一种 TLB：每个入口内容被扩大为 2 倍以容纳 2 个各自映射独立物理页的连续的 VPN。这种成对的入口仅增添了很少的硬件逻辑但却加倍了 TLB 可以装入的映射页，避免了对 TLB 的设计进行大幅度的调整。

您可以看到为何被称为“全相连”，这强调了所有的键值实际上是并行对输入值进行比较的。

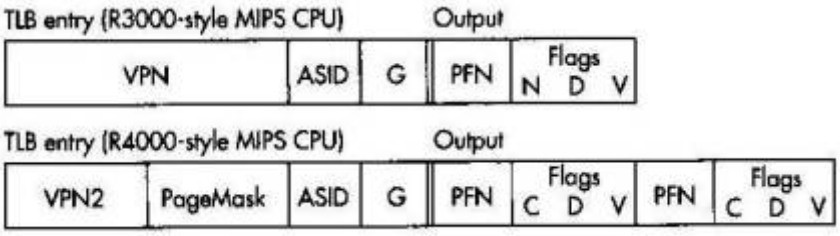


图 6.3

TLB 的入口如图 6-3 中所示（您可以在后面的 6.5 节中找到其详细的编程信息）。TLB 的键值包含了以下内容：

VPN：虚地址的高位（即虚页地址）。在双入口 TLB 内被称作 VPN2，这是为了强调如果每个物理页都是 4KB，那么一个虚地址将会去掉低位（这些低位用来选择左边的输出域或右边的输出域）来进行比较以选出一对入口。

PageMask：这只有近来的 CPU 才有。它用来控制使用虚地址的多少位来跟 VPN 进行比较并决定多少位被通过后加入实地址；使用越少的位达成的匹配映射的空间越大。MIPS CPU 能够设置一个入口映射最大达 16MB 的内存。在使用各种页大小的情况下，都是用被掩位中的最高位来选中奇数号或偶数号入口。

ASID：标记这个转译过程属于某一个特定进程，因此除非 CPU 的当前 ASID 与之相吻合，否则匹配是不会成功的。“C”这一位如果被置起为 1，则关闭 ASID 的匹配，这标志着本转译可以在所有的进程空间内进行(因此地址映射中的这一部分是被所有空间共享的)，ASID 位在早期 CPU 中为 6 位长度，而在近期的 CPU 中则为 8 位。
TLB 的输出部将会给出物理页号和一批数量不大但足够使用的标志位。

物理页号（PFN)：32 位的 CPU 仅仅有一个 N（noncacheable，不可缓存）位——0 表示可以缓存，1 表示不可缓存。
而 64 位 CPU 则提供了一个 3 位的 C 域来表示一个更大范围的取值，可以来通知多处理器硬件在访问与其他处理器共享的页面数据时遵循何种协议。不具备硬件上的快存一致性协议特征的 64 位 CPU 保留了这样的 TLB 入口的格式；在所有 R4000 类型的 CPU 中只有 2 个取值来表示可被缓存（3）或者不可缓存（2），后者在 R4000 类型 CPU 中为标准默认值。现代的嵌入式 CPU 可以使用不同的取值来选择不同的快存策略：一对是写透（write through）与写回（write back），另一对为写分配（write allocate）与写不命中的非缓存策略（uncached write on miss)。详情请查阅您的 CPU 用户手册。

写控制位（D)：置为 1 则允许数据写入相应的页。“D”来源于其全称“dirty”；请在 6.8 节中寻找原因。

Valid 位（V）:如果是 0，则相应的入口是不可使用（不可用于地址转译）的。看上去好像这没什么意义：既然我们不想转译机制工作，干嘛要把相应的纪录存入 TLB 呢？这是因为进行重装入动作的软件为了优化速度，不希望去检查特例。当需要在程序能够使用一个内存常驻页表中的页之前对该页进行某些处理时，相应的入口可以先被标记为无效（invalid)。当 TLB 重装入完成后，这会引发一个不同类型的陷入，调用特殊的程序来进行处理，而不必在每一个重装入事件中都进行测试。

现在转译一个地址就变得很简单了，我们可以把上面所描述的过程充实如下：

CPU 产生一个程序地址:无论是取指令,还是装入和写回数据,只要这些不处于 MIPS 地址空间中特殊的未映射区间时就会进行转译。
低 12 位被分离开来，剩下的处于 EntryHi 的 VPN 和 ASID 相拼作为 TLB 的键值，TLB 入口中的 PageMask 位与 C 位对这个值有修改效果。

TLB 进行键值匹配：匹配成功的入口被选出。PFN 被粘贴在程序地址的低位之前以产生一个完整的物理地址。

地址有效吗？V 位和 D 位被参考。如果地址为无效或者正试图写一个 D 位为 0 的页，CPU 产生一个陷入动作。在所有的转译陷入操作中 BadVaddr 寄存器都会装入引发陷入的程序地址；而在任何的 TLB 异常中，TLB 的 EntryHi 寄存器将被预先装入引发陷入的程序地址的 VPN。
请不要在 TLB 不命中处理以外的情况下使用便利寄存器 Context(64 位 CPU 中为 XContext)，在其他时候它们可能用来追踪 BadVaddr 之类的东西，或者干脆不用，两者都是允许的。

是否被缓存？如果 C 位被置起,那么 CPU 就到快存中去查找物理地址中的数据拷贝；假如数据不在那里，那么就会到内存中去取并且留一份拷贝到快存中。对于 C 位未被置起的入口，CPU 既不查找快存也不把相应地址的数据装入快存。

当然，TLB 的入口数量只能帮助你转译相对较小的程序地址空间——大致在几百 KB 左右。对大多数系统来说这远远不够，TLB 也几乎总是作为一个软件来维护的面向一个更加巨大的转译集合的快存来使用。
当 TLB 中一次地址查找失败后，会陷入一个 TLB 重装入异常。系统软件需要做如下工作：

判断是否存在一个正确的转译；如果不存在，这个陷入会被派发到用于处理地址错误的程序中去。

假如存在一个正确的转译，那么创建一个用于实现转译的 TLB 入口。

假如 TLB 已经装满（在运行中的系统中它基本上也总是满的），软件要选择一个可以丢弃的入口。

软件把新的入口内容填入 TLB。

请翻阅第 6.7 节来看一下这些是如何处理的，但这里请注意，尽管有特殊的 CPU 特性来帮助实现这一类重装入动作，软件还是可以用任意方式来进行 TLB 的重装入。

6.3 MMU 的寄存器

现在我们终于可以把这一套自顶向下的方法收起来了，让我们深入到 MIPS 的实现细节中去吧。我希望您有足够的知识背景来挖掘文中的内容；一旦我们陈述了细节内容，我们就会展示一下相应的器件是如何工作的。

就像 MIPS CPU 中所有其它部分一样，MMU 的控制是受少量的额外指令和一小部分协处理器中 0 号集里的寄存器影响的。表 6.1 列出了控制寄存器，我们还需要用到 6.4 节中所使用的指令。

寄存器助记符	CP0 寄存器号	描述
EntryHi EntryLo/ EntryLo0 EntryLo1 PageMask	10 2 3 5	这些寄存器共同装下了一个 TLB 入口所需的所有信息。所有对 TLB 的读写操作都必须通过它们。EntryHi 含有 VPN 和 ASID；EntryLo 含有 PFN 以及一些标志。 事实上 EntryHi（ASID）域有 2 个职责，因为它还负责了记住当前活跃的 ASID。 在某些 CPU 里（至今为止还都是那些 64 位 CPU）每个入口会映射 2 个连续的 VPN 到不同的物理页，独立由被称为 EntryLo0 和 EntryLo1 的 2 个寄存器所指明。 EntryHi 在 64 位 CPU 中增加到 64 位，但是为不需要用到长地址的软件保留了 32 位的格式。 PageMask 可以用来创建能映射超过 4KB 的页的入口；参见 6.3.1 节。
Index	0	用来在使用适当的指令时决定读或写哪一个 TLB 入口
Random	1	这个伪随机值（实际上是一个自由计数的计数器）用来让 tlbwr 写入新的 TLB 入口到一个随机选择的位置。为那些使用随机替换的软件在陷入 TLB 重装入异常时的处理节省了时间（或许也没有其他合适的替代方法）。
Context XContext	4 20	这些是很有用的寄存器，用来加速 TLB 重装入的过程。它们的高位可读写，低位从不可转译的 VPN 中得来。 寄存器的域这样布置使得如果您使用了合适的内存转译纪录的内存拷贝的安排方式，那么紧跟在 TLB 重装入陷入后 Context 会装有一个指向用来映射触发异常地址的页表纪录的指针。参见 6.3.5 节。 XContext 在处理超过 32 位有效地址时做了相同的工作；由于受操作结果的数据结构的大小影响，对 Context 寄存器格式的直接扩展是行不通的。某些 64 位 CPU 上

		的软件也乐意使用 32 位虚地址，但是当这些不够的时候 64 位 CPU 装备了“模式位”SR（UX）和 SR（KX），它们置起后能导致一个替代的对 TLB 重装入处理方法被调用；同时这个处理方法可以使用 XContext 来支持一个更加巨大且可管理的页表格式。

表 6.1

6.3.1 EntryHi，EntryLo 和 PageMask 寄存器

图 6.4 展示了这些寄存器，这些也是程序员对 TLB 仅有的可见部分，它们的设计也被一起进行了最佳化的考虑。

EntryHi 的内容域解释如下：

VPN，VPN2（虚页号）：这些是一个程序地址的高位部分（忽略 0-11 位的剩余部分）。VPN2 把 12 位也忽略掉，因为相应 TLB 的入口会映射一对 4KB 虚页。在重装入异常产生后这个域会自动用来匹配无法进行转译的程序地址。当你想写入一个不同的 TLB 入口时，或想要检查一下 TLB 内容，你必须要手动来进行设置。64 位的系统（迄今为止）并不真正意义上支持像上面所隐含表示的一样巨大的虚地址空间。VPN2 在 R4X00 CPU 中事实上是一个 27 位的域，以适应 40 位的程序地址空间。VPN2 的高位**必须**被写为全 1 或者全 0，匹配相应的 EntryLo 寄存器的最高一位；同时，高位全为 1 代表访问的是内核地址空间，否则高位全为 0。

如果您只使用 32 位的指令集这些是自动完成的，因为当您在这种方式下工作时所有的寄存器值都包含对 32 位数的 64 位符号扩展。

ASID（地址空间标识）：这通常用来保存操作系统所看到的当前地址空间的标示。异常不会改变它，因此在一个重装入异常发生后，它依然为当前运行的进程保存着正确的标识信息。

绝大部分软件系统会蓄意把这个域写入当前的地址空间。然而，当你使用 tlbr 来检视 TLB 入口时必须要小心；这个操作会覆盖写入整个 EntryHi，因此在这之后您必须恢复写入正确的当前 ASID。

R：这是地址空间标识。您可以仅仅把它看为 EntryHi（VPN2）的另一部分地址位；实际上它也只是 64 位 MIPS 虚地址的最高位。不过，如果您能回忆起 64 位 MIPS 的扩展内存映射（参见 2.8 节中的图 2.2），您可以发现这些高位把内存区域区分出不同的访问权限。同时，它们又与 VPN2 的高位有所不同，因为实际上它们可以使用不同的取值——EntryHi 的那部分在实现中定义的高位必须是全 1 或者全 0。

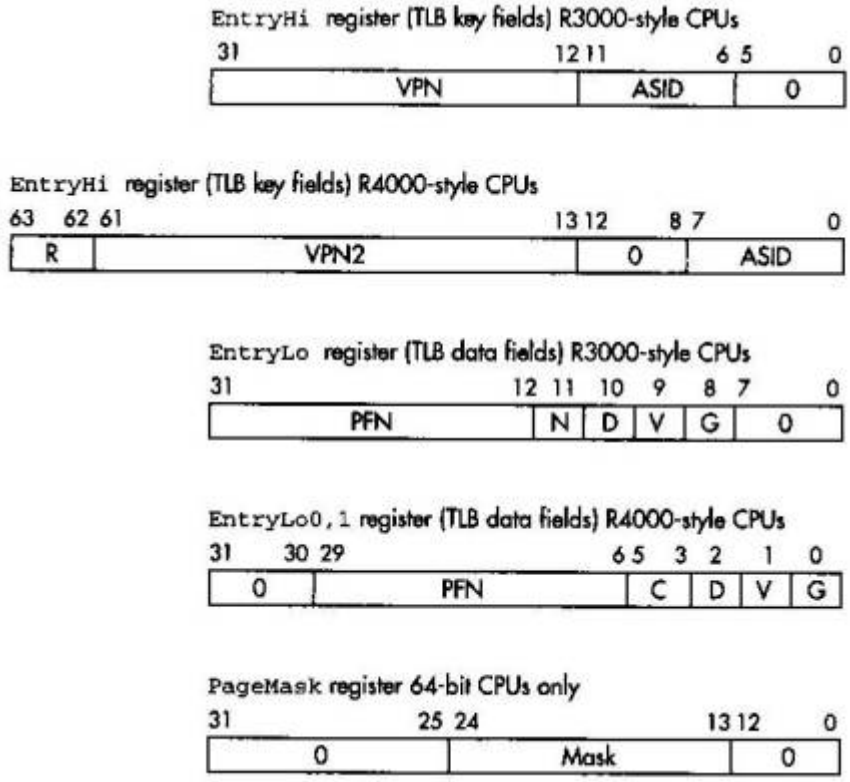


图 6.4

EntryLo 的域解释如下：

PFN：这是相应 EntryHi 的 VPN 所转译出的物理地址的高位。

N（不可缓存标志）：置为 0 允许相应地址的数据访问被快存所缓存，1 则不可缓存。

C：对 R4000 及以后的 CPU 而言，内存访问有更丰富的快存策略可供选择，这被编码在一个 3 位长度的域中。不过除了“不可缓存”（2）和“多处理器下非信号缓存”（3）之外的其他取值在具有快存一致性能力的多处理器和更晚出现的嵌入 CPU 中都有不同的使用方式。

D（dirty 位）：这用来作为一个“写允许”位。置为 1 表示允许写回，置为 0 则导致任何使用本转译来进行的写操作被陷入。请参见 6.8 节中对术语“dirty”的解释。

V（valid 位）：如果置为 0，那么任何一个匹配本入口的地址访问将导致一个异常。这既可以用来表示一个页当前不可访问（在一个纯虚地址系统中），也可以用来表示位于 EntryLo 的一对转译都不可用。

G（global 位）：当一个 TLB 入口的 G 位被置起时，TLB 入口将只使用 VPN 域来进行匹配，而不顾 TLB 入口的 ASID 域是否与 EntryHi 内的值一致。这就使得我们可以实现一部分地址空间在所有进程中共享，而不需要增加额外的页表。

0 域：这些域一直都为 0，但与许多保留域（reserved field）不同，它们不需要被写为 0（写入数据也不会有任何事发生）。这很重要；这意味着在重装入 TLB 时用于产生 EntryLo 的内存常驻数据可以在这些域里保留一些软件来解释的数据，TLB 硬件会忽略这些字段而不需要浪费宝贵的 CPU 时间来把这些位屏蔽掉。

PageMask 寄存器至今为止已经在所有的 64 位 CPU 内实现。当前的 mask 域在 TLB 入口创建时被拷贝进去，置为 1 的位会起到导致虚地址的对应位在匹配 TLB 入口时被忽略的效果（并导致那一位不做修改地传到最终物理地址上），这有效的帮助了匹配一个更大容量的页面。

地址中被屏蔽的位也同样被直接拷贝到物理地址上。
没有一款 MIPS CPU 允许在 Mask 中使用任意的位排列样式。绝大多数都是允许页大小从 4KB 到 16MB 之间，页大小是以 4 倍关系递增的：

PageMask bits			Page size
24–21	20–17	16–13	
0000	0000	0000	4KB
0000	0000	0011	16KB
0000	0000	1111	64KB
0000	0011	1111	256KB
0000	1111	1111	1MB
0011	1111	1111	4MB
1111	1111	1111	16MB

NEC 的 Vr4200 CPU 只支持 4KB 和 16MB 两种页，不过相应编码都是在硬件中固定好的。

6.3.2 Index 寄存器

Index 寄存器是当你有意去写一个特定的 TLB 入口时用来指定那个入口的，还可以用来在你使用 tlbp 查找某个转译后返回相应的 TLB 索引。
图 6.5 中可以看到，Index 不仅仅是一个数字。P 域在 tlbp 指令查找一个合法转译失败后被置起；由于它是寄存器的最高位，因此这个结果看起来就像是产生了一个 32 位的负数，检查起来是很容易的。
请注意，早期 MIPS CPU 各域具有不同位置，而且有效位只有 6 个（最多定位 64 个 TLB 入口）。

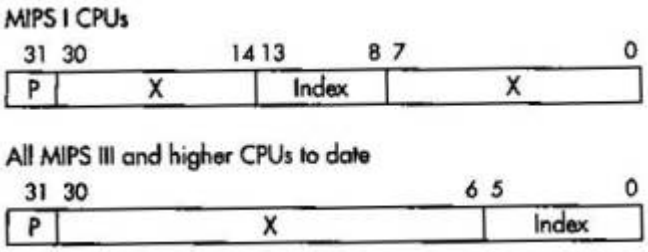


图 6.5

6.3.3 Random 寄存器

Random 寄存器在 TLB 里保有一个在 CPU 执行每条指令时进行计数（是递减的，如果这个特征对您来说重要的话）而得的索引，它在写 TLB 入口的 tlbwr 指令执行时作为 TLB 的索引，以此支持写 TLB 入口的随机替换策略。
通常情况下您永远不需要去读写 Random 寄存器（图 6.6 所示），不过在诊断过程中它可能是有用的。我们可能会期望在系统重启（reset）时把硬件的 Random 域置为最大值——相当于选择最大序号的 TLB 入口，并且每个时钟周期它都会递减，直到达到某个基值（floor value），然后数值回卷（wrap back）变为 63，重新开始递减。
因此从 0 号到小于基值的 TLB 入口不受随机替换的影响，操作系统可以使用这些槽作为永久的转译入口——在 MIPS 上操作系统中这被称作“绑定的”（wired）。

在早期 CPU 中基值固定为 8，不过对这个常数设置有点霸道的做法招致了一些抱怨，因此 64 位的 CPU 引入了 Wired 寄存器，允许更改基值，因而也就可以让 Random 寄存器的取值范围有所变化。

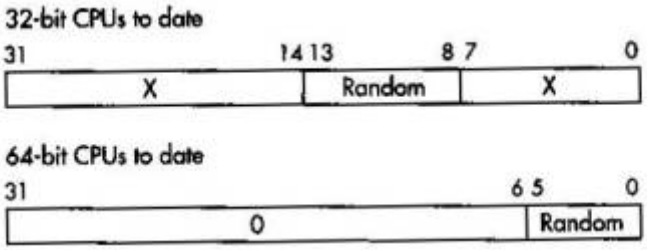


图 6.6

6.3.4 Wired 寄存器

这只是一个数值，不过里面的绑定值超过 TLB 最高索引时的就不会带来有意义的效果。当您写入 Wired 寄存器后，Radom 寄存器就会自动重置为指向 TLB 的最大序号入口。

6.3.3 Context 寄存器及 XContext 寄存器

当某个转译不在 TLB 中而导致 CPU 引发一个异常时，不能完成这次转译的虚地址就会存入 BadVaddr 寄存器，而且 VPN（TLB 所关心的部分）也已经存入 EntryHi 寄存器。很明显，这已经做了足够多的工作，然而为了加速异常的处理过程，Context 和 XContext 寄存器会以某种格式把相同的信息重新打包一下，使得形成一个立刻可以用于内存页表的指针值。

图 6.7 展示了这些寄存器，各个域的描述如下：

PTEBase：这是一个存储您写入的定位信息的部分。为了实现“标准”的重装入处理过程，这应该是内存常驻页表起始地址的高位。起始地址必须选择一个第 20 位及以下的位为 0 的地址，因为 Context 寄存器会用来做一个“逻辑或”运算，而不是做加法。这就限制了内存保留的页表必须在虚地址上以一个 1MB（疑点，照图中看来似乎应该是 2MB）边界的地址起始——这大概也不是什么了不起的麻烦。

Bad VPN / Bad VPN2：在一个地址异常之后这里会装入相应地址的高位（疑点，在图 6.4 中 VPN 是 20 位长，而图 6.7 中 Bad VPN 是 19 位长），与 BadVaddr 寄存器中的高位完全一样。为什么是 VPN2？如果您 CPU 的 TLB 成对存放入口，那么地址的第 12 位就不属于 TLB 键值域。

VPN 和 VPN2 的值会事先进行左移位，用来预先计算好一个入口长度大于 1 字节的结构指针。32 位 CPU 中 2 位的移位允许 4 字节长度的入口，这足以装入充分多的信息以填充用于构造 TLB 入口另一半的 EntryLo 寄存器。64 位 CPU 不仅仅有 64 位的 EntryLo0 和 EntryLo1 寄存器，还必须各有 2 个，因为每个 TLB 入口映射 2 个页；因此页表就必须期望有 16 字节长的入口，VPN 就会左移 4 位。

0 域：这些域的值永远为 0

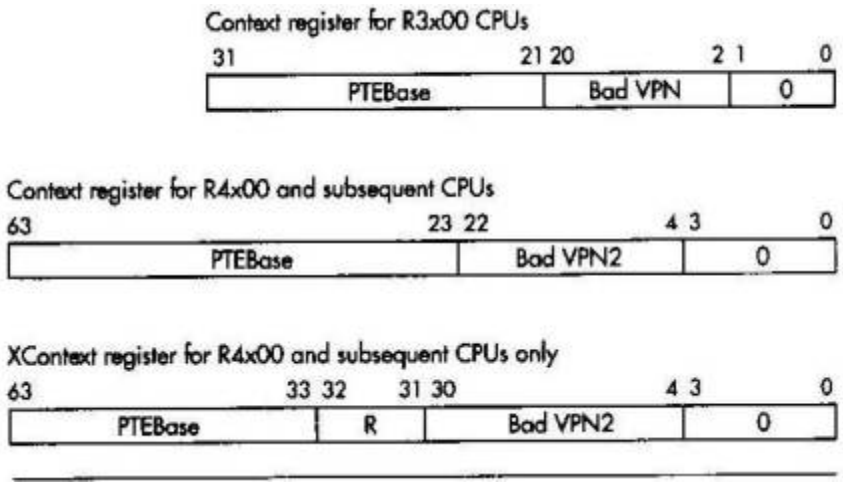


图 6.7

6.4 MMU 的控制指令

下面 2 条指令

```
tlbr        # 根据 index 寄存器读 TLB 入口
tlbwi   # 根据 index 寄存器写 TLB 入口
```

用来在 index 寄存器所选出的 TLB 入口和 EntryHi 与 EntryLo 寄存器中交换 MMU 相关数据。您不会经常需要读一个 TLB 入口；当您这样做时，请记住您会覆写 EntryHi（ASID）域，这被设为与当前运行进程的地址映射相关。所以请把它写回原值。

指令

```
tlbwr   # 根据 Random 寄存器写入 TLB 入口
```

会拷贝 EntryHi（包括 ASID 域），EntryLo 和 PageMask 寄存器的内容到 Random 寄存器指明的 TLB 入口中去——如果您使用随机替换策略这将节省一些时间。实际使用中，tlbwr 会在一个 TLB 重装入异常处理中用来写入一个新的 TLB 入口，其他任何情形下则使用 tlbwi。

指令

```
tlbp        # TLB 查找
```

在 TLB 中查找虚页号和 ASID 跟 EntryHi 寄存器中内容相匹配的入口，并把相应入口的索引值存入 Index 寄存器。如果匹配失败，Index 寄存器的 p 位被置起——这使得产生一个负数值，很容易检测。

如果超过一个入口匹配成功，任何事情都可能发生。这是个严重错误，正常情形下是永远不会出现的。

注意 tlbp 指令并不从 TLB 中取数据；您必须在其后运行 tlbr（根据索引读取 TLB 内容）指令来做这件事.

TLB 内部是流水化的，这些用于管理和检查的指令表面上掩盖了这一点。许多实现中要求 tlbp 指令后不能紧跟存取存储器操作。

6.5 TLB 相关编程

TLB 入口是通过向 EntryHi 和 EntryLo 寄存器填入所需的域并通过 tlbwr 或 tlbwi 指令把这个入口拷贝入 TLB 适当位置而设置起来的。

当您在处理一个 TLB 重装入异常时，您会发现 EntryHi 的内容已经为您设置好了。请当心不要创建 2 个匹配同一个“地址 / ASID”组合的入口。如果 TLB 包含重复的入口，

试图转译这个地址或查找这个地址潜在上有破坏 CPU 芯片的可能。有些 CPU 在这些情形下通过关闭 TLB 来保护自己，表现为 SR（TS）位被置起。从此时开始直到硬件重启前，TLB 不会进行任何匹配。
系统软件通常根本不需要读 TLB 入口。不过如果您想读它们，可以使用 tlbp 指令来查找一个装有特定地址的 TLB 入口并看到结果反映在 Index 寄存器。不要忘记先保存好 EntryHi 寄存器并在查找之后重新恢复它，因为它的 ASID 域可能是很重要的。
您可以使用 tlbr 来读取 TLB 入口中的内容到 EntryHi 和 EntryLo 中去。
您可以发现 CPU 的文档中为了进行对指令和数据地址的转译而把 TLB 结构划分成 ITLB 和 DTLB；这些都是微小的硬件控制快存，它们的操作对软件是完全透明的。

6.5.1 重装入是如何产生的

当程序访问任何一个转译好的地址区域（通常在一个被保护的操作系统下，kuseg 为应用程序使用，而 kdeg2 对应内核特权的映射）时，如果 TLB 中没有相应的转译纪录，CPU 就会引发一个 TLB 重装入异常。
TLB 只能映射现代的服务器或工作站物理内存空间的一小部分。规模庞大的操作系统会维护某种包含大量页转译信息的内存常驻页表，并使用 TLB 来作为最近所进行的转译的快存。大多情况下页表是一个可以立即使用的 TLB 入口的数组，您可以使用 Context 寄存器作为一个指向它的指针。
由于 MIPS 系统通常把操作系统内核置于非转译内存区 kseg0，因此通常情形下是一个用户级的程序需要转译一个 kuseg 地址。有几个硬件特性就是为了加速这种通常情形下异常处理而设计的。首先，这些重装入异常会以向量形式置于不产生其他异常的内存低地址。第二，设计上使用的一系列精巧的小花招使得内存常驻页表被分配在内核虚地址（kseg2 区域或者 64 位系统中的相应部分）中，这样物理内存空间就不需要存放页表中用来映射进程地址空间里的“空洞”部分。
最后，Context 和 XContext 寄存器可以用来立即对内存常驻页表中的正确入口进行访问。我们会在第 6.7 节详细剖析这个过程。但是在我们深入探讨之前，我们应该注意所有这些特性都是“非强制”的。在一个小的系统中，TLB 可以用来创建一个固定的或很少更改的用以进行从程序（虚地址）到实地址转换的转译器件，这些情况中 TLB 甚至不需要作为一个快存来使用。
甚至在某些较大的虚存操作系统在 MIPS 上的实现中也都并不使用“标准”的页表。可移植的 NetBSD 内核的早期版本中组织了一个相对庞大的软件管理的转译信息的二级缓存，在通常的重装入代码中会进行查找；访问其转译不在二级缓存中的页面是很罕见的，这会转交给一个相对重量级的由 c 语言编写的处理过程，并从一个机器无关的页表中提取相关信息。

6.5.2 使用 ASID

我们通过一个专门的 ASID 配置和把 EntryLo 的 G 位设为 0 来建立 TLB 入口，除非 CPU 的 EntryHi（ASID）寄存器域与 TLB 入口的相应值相符，那些入口是永远不会与一个程序地址完全匹配的。这就允许你同时映射 64 或 256 个不同的地址空间，而不需要在进程切换时完全清除 TLB。如果您不使用 ASID，那么您就必须要遍历整个 TLB 并取消所有您不使用 ASID 的地址空间的映射。

6.5.3 Random 寄存器与被绑定入口

硬件并不提供给您找出最近最常使用的 TLB 入口的手段。当您把 TLB 当作一个快存来使用并且需要装入一个新的映射时，唯一实用的策略就是随机替换一个入口。CPU 通过维护一个在每个处理器周期都进行计数（实际中是递减的）的 Random 寄存器，使得做到这点十分容易。随机替换听起来效率低的吓人；您可能恰恰牺牲了那个近来最频繁使用的入口，而那个入口几乎在很短时间内又有访问需要。但事实上当您拥有可观数量的可供选择的牺牲品时，这种情况并不会经常发生以致于带来真正的麻烦，况且大多数 MIPS 上的操作系统也至少留有 40 个可供牺牲。

不过，让某些 TLB 入口被确保留下来直到您主动选择将其移除常常也是很有用的。这对于映射那些您确知会经常使用的页面会很有好处，不过事实上这也非常重要，因为它们使您能够映射一些页并且可以严格确保在这些页的访问上不会产生任何重装入异常。

这些稳定的 TLB 入口被形容为“绑定的”：在 R3000 CPU 中它们由 TLB 的 0 到 7 号入口组成，而在 R4X00 以及后续的 CPU 中入口号的范围可以是从 0 到任何一个您编程写入 Wired 寄存器的值。TLB 本身并不对这些入口做什么特殊处理；魔力来源于 Random 寄存器，它永远不会取从 0 到“绑定值减 1”之间的值；它直接从“绑定值减 1”开始到它的最大值之间循环变化。因此通常的随机替换就不会影响序号到从 0 到“绑定值减 1”之间的 TLB 入口，这些入口被写入后直到明确被移除前都会一直存在。

6.6 内存转译：设置

下面的代码段用来初始化 TLB 以确保它不会对任何位于 kuseg 与 kseg2 的地址进行匹配。我们分别对 R3000 与 R4000 类型的 TLB 完成了这段初始化。下面就是一个简单的为 R3000 或相似 CPU 的 TLB 所进行的初始化动作：

```
#include <mips/r3kc0.h>
LEAF(mips_init_tlb)
mfc0 t0,    C0_ENTRYHI              # 保存 ASID
mtc0 zero, C0_ENTRYLO               # tlblo = valid
li  a1, NTLBID<<TLBIDX_SHIFT        # 索引
li  a0, KSEG1_BASE              # tlbhi = 不可能出现的 VPN
.set noreorder
1:  subu        a1, 1<<TLBIDX_SHIFT
        mtc0    a0, C0_ENTRYHI
    mtc0    a1, C0_INDEX
        addu        a0, 0x1000              # 增长 VPN，使所有入口都不同
    bnes        a1, 1b
        tlbwi                           # 在跳转的 delay slot 中
    .set reorder

        mtc0    t0, C0_ENTRYHI          # 恢复 ASID
    j       ra
END(mips_init_tlb)
```

下面是一个 R4000 或类似 CPU 的 TLB 初始化的简单例子：

```
#include <mips/r4kc0.h>
```

```
LEAF(mips_init_tlb)
dmfc0 t0, C0_ENTRYHI                # 保存 ASID
li  a1, NTLBID                      # 从 TLB 的顶部加 1 开始
li  a0,KSEG1_BASE                   # tlbhi = 不可能出现的 VPN
mtc0 zero, C0_ENTRYLO0              # 0 是非合法值
mtc0 zero, C0_ENTRYLO1

1:  subu        a1, 1
        dmtc0   a0, C0_ENTRYHI
    dmtc0   a1, C0_INDEX
    addu        a0, 0x2000              # 增长 VPN，使所有入口都不同
    tlbwi
    bnes        a1, 1b

.set noreorder
nop                             # tlbwi 后来会使用 entryhi
    dmtc0   t0, C0_ENTRYHI          # 恢复 ASID
        .set reorder

        j       ra
END(mips_init_tlb)
```

让我们看一下 TLB 的初始化过程。

两个程序都从 TLB 的顶部开始（常数 NTLBID 可以在包含的头文件里，在算法里称为 r3kc0.h 或 r4kc0.h）

EntryLo0 和 EntryLo1 中的 0 值意味着任何转译都不是合法的，不过这本身还不足以避免重复入口的麻烦。

注意 R3000 版本里的 Index 拥有一个需要移位的域，因此我们不能仅仅对其加 1。

存放在每个入口中的 VPN 就是 kseg1 区域中的页的相应部分，这些被定义为非转译地址并且永远不进行查找。不过即使如此，我们依然保证所有的 VPN 是不相同的。

6.7 TLB 异常代码示例

这段程序实现了毫无疑问是MIPS的体系结构设计师为UNIX类型的操作系统中用户地址而设计的转译机制。它依赖于为每个地址空间在内存中建立一个页表的机制。页表由入口的线性数组所构成，以 VPN 为索引，VPN 的格式与 EntryLo 寄存器里的位域相同。R3000 类型的单入口 TLB 需要给每个入口一个字的长度，而 R4000 类型的成对的 TLB 则需要 4 个字长（每个入口都由于为了适应更大的地址空间而增长）。

这种设计十分简单，不过却带来了其他的问题。由于用户空间中的每个 4KB 都需要占用一个 4 字节长的表空间，那么 2GB 的用户空间就需要 2MB 的表，这是个令人不安的巨大容量。当然，绝大多数用户地址空间只充满底部（代码和数据）和顶部（一个向下增长的堆栈），在中间有个巨大的间隙。MIPS 所采取的解决方案是受 VAX 体系结构启发而来的，那就是把页表本身也放在 kseg2 区域的虚存空间内。这立刻简捷优雅地解决了 2 个问题：

节省了物理内存；由于页表中间未被使用的间隙永远不会被引用，实际上就没有物理内存被分配给那些入口。

这为上下文切换时重新映射一个新的用户页表提供了一个简单的机制，使得不需要必须在操作系统中找到足够的虚地址来立刻映射所有页表。您仅仅需要更改一下 ASID 的值，那么 kseg2 所指向的页表现在就自动重新映射到正确的页表。这简直是魔法！

当然，这看起来也导致陷入了一个致命的恶性循环，一个 TLB 重装入所需的东西（向 kseg2 装入页表映射）需要另一个 TLB 重装入来提供。我们同样可以解决这个问题：

并不是所有重装入异常中都会调用快速的 TLB 重装入程序；一个对页表的嵌套 TLB 不命中会被派发到一个通用的异常处理入口点。

提供一套受限的允许我们从用户的 TLB 不命中异常处理中来处理嵌套异常（内核 TLB 不命中）的机制。我们会在下面使用独立的范例来讨论这个，因为 R4X00 及后续的 64 位 CPU 使用了一些不同于 R2000 和 32 位 CPU 的小花招。

MIPS 体系结构通过 Context 寄存器（或者为 64 位 CPU 的扩展地址而设的 XContext 寄存器）的格式来支持这种线性页表。

如果您令页表在 1MB 边界开始并且用页表起始地址的高位来设置 Context 的 PTEBase 域，那么随着用户的重装入异常处理，Context 寄存器将会保有您需要重装入的入口地址，而不需要更多的计算。

6.7.1 32 位 R3000 类型的用户 TLB 不命中异常处理

32 位 CPU 拥有一个用来处理用户可访问地址的 TLB 不命中的异常处理入口点。由地址访问限制引起的 TLB 不命中被送到标准的异常处理入口。这里是一段典型的 32 位 CPU 的重装入程序：

```
    .set noreorder
    .set noat
TLBmissR3K:
    mfc0    k1, C0_CONTEXT      # (1)
    mfc0    k0, C0_EPC          # (2)
    lw      k1, 0(k1)               # (3)
    nop                         # (4)
    mtc0    k1, C0_ENTRYLO      # (5)
    nop                         # (6)
    tlbwr                       # (7)
    jr      k0                  # (8)
    rfe                         # (9)
    .set at
    .set reorder
```

UTLB 的不命中异常是一小段很底层的代码，因此.set noreorder 告诉汇编器（assembler）我们会负起令这段代码序列在 CPU 的流水线上正常执行的责任，而不需要汇编器来关心这个。.set noat 告诉汇编器不允许使用 at 寄存器来综合指令——这很重要，因为我们是从一个任意性很强的异常中进入，at 寄存器里还有未被保存过的用户状态。

k0 和 k1 是确定交由我们来使用的，因此我们不用担心我们在里面所覆写掉的内容。

下面是对这段代码的逐行分析：

1) Context 寄存器是页表的指针。mfc0 指令并不对 MIPS 的 5 级流水线立即产生作用，因此我们在第(3)行之前还无法使用这个指针。

2) 某个时候我们会需要用到返回的地址；现在就在 delay slot 中做这件事。在页表本身也遭受到 TLB 不命中异常的情况下也需要做这个。

3) 运行到这一点时页表的入口地址本身或许在 TLB 中也没有合法的转译入口，这种情况下我们会在这里产生另一次异常。我们会在后面来处理这种情况。

4) 取存指令需要花费 2 个时钟周期,因此在能够使用页表的值之前我们需要等待一下。

5) 把新的值存入 EntryLo。EntryHi（VPN）会由硬件在 TLB 不命中异常产生时自动设置，值代表未命中的地址。EntryHi 中依然保留着我们先前存入的 ASID 值，假定操作系统曾经做过一次进程上下文切换。

6) 等待新的值到达 EntryLo。

7) 向当前的 Random 寄存器恰好指向的 TLB 的位置写入内容，丢弃原先值… …管它哪个位置。不过没关系，这也正是随机替换的有趣之处呢。

8) 返回用户程序，不过每次跳转中 delay slot 里的指令会在我们跳到某个地址之前先去执行… …

9) rfe 指令用来恢复在 SR 寄存器中所保存的异常产生之前的 CPU 状态。

这样我们使用了 9 条指令然后回到遭受 TLB 不命中的程序中去。最大的开销是在从页表中取数据时数据快存不命中而产生的。

不过我们保证过向您解释如果您不太走运地碰上页表入口地址在 TLB 中没有转译入口时会发生些什么。

有一点是不成问题的：像这样的双份转译失败并不常见，所以我们不必特别担忧效率问题。为页表（在特权地址空间内）的 TLB 不命中实现一套重量级的通用异常处理是可行的。

MIPS 的异常实际上只做三件事情：

修改 SR 来关闭中断并令 CPU 转入内核模式

在 EPC 中保存重新开始的位置

被引导到异常处理程序处开始执行

为了允许第二次异常的发生并且能正确返回最初的程序中去，我们需要避免丢失原始的返回地址并且能够恢复 SR 成为上次异常的值。

没有硬件来支持保存返回地址，不过在上面您可以看到，异常处理程序已经把它保存在 k0 中；我们只需要确保通用的异常处理像大部分其他寄存器一样对待 k0，并且保护它的值。

状态寄存器就复杂一些，不过在这里硬件确实会有些支持。起作用的 2 个标志位分别是中断使能位 SR（IEc）和内核模式标志位 SR（KUc）。状态寄存器实际上为这 2 个位提供了一个三层的栈，在异常产生时这 2 个位被压栈，而在异常结束的 rfe 指令时被弹栈，如图 6.8 中所示。

因为 SR（Kux, IEx）构建了一个三层深的栈，即使在第二次异常产生，用户程序的值也依然安全的保存在 SR（KUo, IEo）中，预备着被弹回正确的位置。

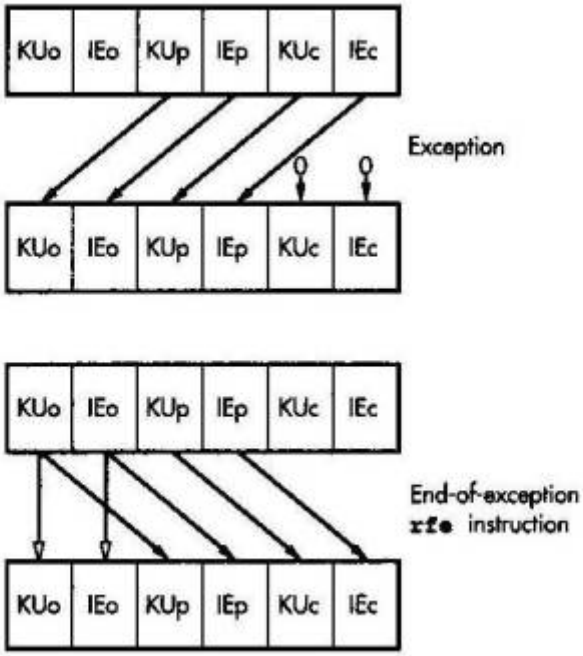


图 6.8

6.7.2 R4x00 类型 CPU 的 TLB 不命中异常处理

R4000 和后来的 CPU 使用成对入口的 TLB，处理双份异常的条件有所不同，这导致产生了这份不同的处理代码。

R4000 拥有 2 个特殊的异常入口点。与 R3000 相同位置的入口只用来处理 32 位地址空间的转译；另一个入口则被提供给标记为使用可处理更大地址空间的 64 位指针的程序。

R4000 的状态寄存器有三个域，SR（UX），SR（SX）和 SR（KX），用来根据发生失败转译时 CPU 所处的特权等级来选择使用哪个异常处理。

R4000 在决定 TLB 不命中何时使用特殊的处理入口以及何时把它派发到通用的异常处理程序有着不同的标准。除非 R4000 已经在处理一个异常——也就是说，除非 SR（EXL）被置起——，否则它总是会使用特殊的处理入口。处理双份异常时就像上面所说；不过由于内核地址的不命中通常会通过与用户地址不命中相同的 TLB 处理程序，R4000 的页表必须大得足以跨越内核的虚地址（也产生了更大的“空洞”）。

这里是 R4000 的 32 位地址空间 TLB 不命中的处理代码：

```
    .set noreorder
    .set noat
TLBmissR4K:
    dmfc0   k1, C0_CONTEXT      # (1)
    nop                         # (2)
    lw      k0, 0(k1)               # (3)
    lw      k1, 8(k1)               # (4)
    mtc0    k0, C0_ENTRYLO0 # (5)
    mtc0    k1, C0_ENTRYLO1 # (6)
    nop                         # (7)
    tlbwr                       # (8)
    eret                            # (9)
```

```
    .set at
    .set reorder
```

下面是对代码的逐行分析：

(1) 这有点怪，64 位的搬运指令在这里好像并不必要：如果页表像通常一样位于 kseg2，那么 Context 的页表基地址部分会被确保在高位全为 1，因此在这里如果您使用 32 位的 mfc0 指令，k1 寄存器会得到同样的值。

(2,7) 某些 CPU（通常是流水线超过 5 级的 CPU，比如 R4000）在这种位置需要额外的 nop 指令。

(3-6) 在这里入口是成对出现的，不过 EntryLo0 和 EntryLo1 仍然只是 32 位的寄存器。然而，Context 寄存器是为了 16 字节长的页表入口而设立的；这些 CPU 中 EntryLo0 和 EntryLo1 并没有不关心的位，软件程序需要一些页表空间来保留一些仅供软件使用的信息。

这里并不需要 nop 指令，因为我们使用了第 2 个 load 指令，因此在每一对 load 和 mtc0 指令间总是有一条其他指令来隔开它们。

就像以前一样，如果页表入口的地址在 TLB 中没有合法转译，我们可能会碰上另一个异常。这次我们还是在后面讨论这一点。

(7) 这里可能在某些 CPU 上需要一个 nop 指令，在长流水线的 R4000 上您就需要一个。

(8) 这里跟以前讨论的一样，是一个随机替换。

(9) MIPS III 和后续的 CPU 有 eret 指令来从异常返回，并且恢复由于异常而改变的 **SR**（对 MIPS III CPU 而言，一个异常对 **SR** 所做的所有事情就是置一下 **SR（EXL）**位）。

在这些后来出现的 CPU 中，如果出现另一次 TLB 不命中会发生些什么事情呢？像以前一样，第二次不命中会被送到一个通用的异常处理入口中，不过这一次是由于 SR（EXL）被置起而产生的（表示正在处理一个异常）。

产生的结果也与 R3000 有所不同。当 SR（EXL）置起时允许产生第二个异常，不过这不会导致替换异常返回地址寄存器 EPC。

在效果上，内核 TLB 不命中异常导致控制权转入通用异常处理入口，同时把 cause 寄存器和所有的用于指示 TLB 不命中的页表入口地址的寄存器都设置好，然而 EPC 却依然指向最初用户空间导致异常的那条指令。内核页表不命中将会被妥善解决（如果可以的话），然后通用异常处理会返回用户程序。显然，我们还没有做任何事情来处理最初的用户地址空间 TLB 不命中，因此这会立即再产生一次不命中。不过这一次，所需的内核转译内容已经可用，用户的不命中也可以成功的完成。

6.7.3 XTLB 不命中处理

通过设置适当的位（一般就是 **SR（UX）**），TLB 不命中会被送到一个不同的处理向量中去，在那里我们应该有一段用于大地址空间的转译装入的程序。处理代码看起来与先前是相同的，除了使用 XContext 寄存器代替了 Context：

```
    .set noreorder
    .set noat
TLBmissR4K:
    dmfc0   k1, C0_XCONTEXT # (1)
    nop                     # (2)
```

```
lw      k0, 0(k1)               # (3)
lw      k1, 8(k1)               # (4)
mtc0    k0, C0_ENTRYLO0 # (5)
mtc0    k1, C0_ENTRYLO1 # (6)
nop                         # (7)
tlbwr                       # (8)
eret                            # (9)
.set at
.set reorder
```

不过请注意，结果所得的内核虚存中的页表结构会比以前大得多，我们需要对内核虚存映射和转译代码做些细小的改动来适应这点。

6.8 跟踪被修改页（模拟“Dirty”位）

为应用程序提供页面的操作系统常常需要跟踪这个页面自从上次操作系统获取它(可能从磁盘上或网络上）或保存它的拷贝之后有没有被修改过。未被修改（“clean”）的页可以直接被丢弃，因为下次需要它们时可以从文件系统中简单的恢复它们。

在操作系统的说法中那些被修改过的页面被称为“dirty”，操作系统必须密切注意它们，直到应用程序退出或者这些 dirty 页被保存回去而就此清除掉。为了帮助实现这个过程，CISC CPU 通常在内存常驻页表中维护一个位来指示这个页上发生过一个写操作。MIPS CPU 不支持这个特性，甚至在 TLB 入口中也不。页表的 D 位是一个写允许位，当然也是用来标志只读页的。

这里是技巧所在：

当一个可写的页第一次装入内存时，它的页表入口的 D 位并不被改动（也就是让它成为只读的）。

当任意一个尝试对这个页写动作出现时会产生一个陷入；系统软件会识别这是个合法的写动作但是却通过这个事件在常驻页表中设置一个修改位——由于它在 EntryLo（D）的位置，这就允许将来的写操作顺利进行而不会产生一个异常。

您也许希望通过设置 TLB 页表中的 D 位来允许写操作的进行，但是由于 TLB 入口是随机且不可预测的进行替换的，因此这个方法对记住被修改状态是没有帮助的。

6.9 内存转译和 64 位指针

当 MIPS 体系结构被发明时，32 位 CPU 已经出现了一段时日，而且最大的程序数据也已经达到了 100MB——地址空间只剩余 4 位左右可以用来分配。因此我们必须合理使用 32 位空间并且要避免它被肆意挥霍的碎片所侵蚀；这也是为什么应用程序（在用户级运行）会为自己保留 31 位的地址空间。

当 MIPS III 指令集在 1991 年引入 64 位寄存器时它是业界领先的，而且就象我们在 2.7 节所讨论的那样，MIPS 超前了 32 位地址限制所真正产生压力的时候大约 4 到 6 年。双倍的寄存器大小只是必须产生出少量的额外地址位来适应未来的需要；小心对待操作系统潜在的爆炸性增长的数据结构比有效使用所有的地址空间变得更加重要。

由基本的 64 位地址映射而造成的实际地址空间的限制在一段时间内依然是无法解决的；它们允许被映射的用户空间或其他空间不加识别的增长到 61 位地址大小。然而，XContext

（VPN2）域“只有”27 位，这就限制了可映射的用户虚地址为 40 位。这样的话我们如何来实现一个 40 位的用户空间呢？

一个与 XContext 相容的页表拥有 2^{29} 个入口（每个对应 R/VPN2 的一个值，16 字节长）。那就是 8GB 空间，超过了 kseg0，kseg1 和 kseg2 结合起来所能表示的整个空间大小。幸运的是，在 R4x00 以及后继的 CPU 中拥有另一个 2^{40} 字节大小，内核级权限，从 0xC000 0000 0000 0000 起始的映射区域可供使用。这个表中的大部分是空的，这是因为 40 位的用户程序地址空间（对它来说 R == 0）在栈与数据段中间拥有一个极广大的间隙，在特权区域中用到的甚至更少。页表中与这个间隙相关联的部分永远不会被访问到，根本不需要映射到物理内存中去。很明显，使用某种相对紧凑的数据结构来映射内核权限地址是很有用的，不过这就牵扯到了操作系统的设计，超出了这本书的范围。

6.10 MIPS TLB 的日常使用

如果您正在使用一个庞大的操作系统，那么它会使用 TLB 而且您几乎不会看到它。如果不是，您可能会怀疑它是否有用。由于 MIPS TLB 提供了一个相对通用的地址转译服务，在许多方面您或许可能从中受益。

TLB 机制使您可以转换地址（在页粒度上）到任何物理地址上，这样可以重定位程序的地址空间到您机器上任何地址映射中去。如果您的映射需求有限，可以在 TLB 中进行所有的转译，那么就不需要支持 TLB 重装入异常或另外的内存常驻页表。

TLB 也允许您定义某些地址为暂时或永久不可用，因此对那些地址的访问会引起一个异常，引发某些操作系统服务程序。通过使用用户级程序您可以让某些软件仅能访问那些您希望它们可以访问的地址，而且通过在那些地址中使用用户地址空间 ID，您可以有效的管理多个互相不可互相访问的用户程序。您可以写保护某一部分内存。

TLB 的应用不仅仅是这些，这里有个列表来指明了它可应用的范围：

访问不方便的物理地址范围：MIPS 系统的硬件寄存器绝大部分方便地位于实地址的 0-512MB 范围，您可以使用一个 kseg1 区域内的相应指针来访问这个范围。而拥有不在这个期望区域内的硬件的地址，您可以映射它的实地址高位到一个方便的映射区域内，比如 kseg2。这个转译的 TLB 标志位应该设置为不可缓存，不过接下来书写程序时就可以当作这些地址在方便的位置那样。

异常处理程序的内存资源：假设您希望运行一个异常处理程序时不使用保留的 k0/k1 位来保存现场。这样的话，您就可能碰上麻烦，因为通常来说 MIPS CPU 并没有地方能让您保存寄存器时不覆写其他至少一个寄存器。您可以使用 zero 寄存器作为基地址来进行 load 或 store 动作，但是配上一个正的偏移值（offset）后这些地址会位于 kuseg 的第一个 32KB 内，而配上一个负的偏移则会位于 kseg2 的最后一个 32KB 内。不使用 TLB 的话，这些是没什么意义的。有了 TLB 后，您可以映射这个区域的一页或多页到可读写内存区域内，然后使用 0 基址的 store 动作来保存上下文从而挽救了您的异常处理。

非虚存系统中可伸展的栈和堆：甚至在您没有磁盘并且没有支持完整的分页需求时，根据监视应用程序的栈和堆的增长情况来扩大它们也是很有用的。在这个情况下您需要 TLB 来映射栈或堆的地址，然后根据 TLB 不命中事件来决定是否分配更多内存或者应用程序已经失控。

模拟硬件：如果您有一些某些时候可用而某些时候不可用的硬件，那么通过映射区域来访问其寄存器可以在适当配置的系统中直接与硬件建立联系，而访问不可用硬件时就会

导致调用起一个软件的异常处理。

总而言之，拥有所有这些符合大操作系统规范的的精巧设计的 TLB，是一个很有用并且能直接面向编程者的通用资源。

6.11 非 UNIX 操作系统的内存管理

为桌面以外用途所设计的操作系统一般称为实时操作系统（RTOS），这盗用了一个曾用来表示某些真正实时的术语。这一章第一部分所列出的 unix 类操作系统拥有您所能在小操作系统中所有能找到的元素，不过许多 RTOS 会更加简洁。
这个领域很新，并没有事实上的标准。领域内的先锋可能是 Microsoft 的 Windows/CE，而且那个操作系统的内部描述目前还不能自由获取。所以我们将把话题限制在几个方面内。
非桌面系统更倾向于提供一套简单而紧密集成的函数方法;而不需要支持一个变化较大的范围内的程序，这包括第三方或者用户编写的程序，进程的保护不显得特别重要。我们期望较小的操作系统有能许可更多的操作，这是由于应用程序的编写者影响力变得更强。这样做究竟是不是很好目前还并不明确，不过早先的 RTOS 根本就没有保护机制。
作为装入程序的一种方式，调页就很有意义，因为您不需要做装入并不需要的那部分程序的工作。没有磁盘的系统或许并不会换出含有 dirty 数据的页，然而，调页在没有它的情况下依然很有用。
当您正试图理解一个新的内存管理系统，第一件事应该是理解内存映射，包括为软件提供的虚映射和系统的实地址映射。是概念上很简单的虚地址映射使得 unix 的内存管理变得直观。但是定位嵌入式应用的系统通常并不把根基建立在内存管理相关的硬件上，进程的内存映射常常有未映射内存隐藏在其中。用铅笔、纸和耐心就足以解决它了。

Assembler language programming
这一章将告诉你如何阅读并编写 MIPS 体系下的汇编代码。MIPS 汇编代码看上去与实际的代码差异很大，这主要是因为以下原因:
1, MIPS 汇编编译器(assembler)提供了大量的已经预定义的宏指令(extra macro-instruction)。所以编译器的指令集(instruction set)要比 CPU 实际提供的指令集大的多。
2，在 MIPS 汇编代码中有许多伪操作符，放在代码开始和结束的地方，用来预定义常用数据，控制指令排列顺序，以及控制对代码的优化。通常它们被称为“directives”或“pseudops”。
3，实际应用中，汇编代码往往要经过 C 语言预处理器(C preprocessor)的处理后，才被提交给 assembler 进行编译。C 语言预处理器将汇编代码中的宏，用它自己的头文件中的定义进行替换。这可以使汇编代码书写起来稍微方便一点。
在你继续看下去之前，最好先回去温习一下 Chapter-2 的内容,包括低层机器码的构造，数据类型，寻址方式。($：流水线 pipeline 的知识很值得温习一下，主要看一下那些该死的延迟点 delay-slot。)

9.1 A Simple Example
我们仍然采用在 Chapter-8 见过的那个例子: C 库函数 strcmp(1)。这一次我们的演示的重点是: 汇编语法所必需的符号，以及一些人工优化(hand-optimized)并重排序(hand-scheduled)的代码。

```
Int
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 );
}
```

这段代码的运行速度因为以下原因而比较低：

1，　　每个循环都会经过两个条件分支(conditional branch)和两个提取指令(load)，而我们没有在分支延迟点(branch delay-slot)和提取延迟点(load delay-slot)上放置足够的指令($：相当于 cpu 在 delay-slot 处做 nop 动作，从而影响了效率，参见 1.5.5 programmer-visible pipeline effects)。

2，　　每次循环只比较一个字节，使得循环过于频繁而效率低下（$：因为分支(b*)及跳转(j*)指令会造成流水线的刷新，后续指令被失效)。

我们来修改这段代码：首先把循环展开，每次循环比较 2 个字节；把一个 load 指令调整到循环的末尾——这只是一个小技巧，这样我们就可以尽可能的在每个 branch delay-slot 和 load delay-slot 处都放上有效的指令了。

```
Int
Strcmp(char* a0, char* a1)
{
    char t0, t1,t2;

    /*因为第一个 load 被调整到循环的末尾处，所以这里要先取一次值*/
    t0 = a0[0];

    while(1){
        t1 = a1[0];          /*第一个字节*/
        if( t0 == 0 )
            break;
        a0 += 2;                 /*$：branch delay-slot*/
        if( t0 != t1 )
            break;
    /*第 2 个字节，在上面我们已经把 a0 加 2 了，所以这里是[-1]*/
```

```
        t2 = a0[-1];            /*$：branch delay-slot*/
        t1 = a1[1];         /*先不把 a1 加 2，留到下面的 delay-slot 处再加*/
        if( t2 == 0 )
            return t2-t1;       /*下面汇编代码里的标志.t21 处*/
        a1 += 2;                /*$：branch delay-slot*/
        if( t1 != t2 )
            return t2-t1;       /*下面汇编代码里的标志.t21 处*/
        t0 = a0[0];         /*$：branch delay-slot*/
    }
/*下面汇编代码里的标志.t01 处*/
    return ( t0 - t1 );
}
```

ok,现在让我们把这段代码转成汇编来看看。

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF(strcmp)
    .set        nowarn
    .set        noreorder
    lbu     t1, 0(a1);
1:
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)
```

Even without all the scheduling,这里已经有很多有意思的东西了，让我们来看看。

#include
这是个好主意：由 C 语言预处理器 cpp 来对常量进行宏定义，并引入一些预定义的文本宏($：text-subsitution macro，就是上面的 LEAF、END 之类的东西)。上面这个汇编文件就是这样做的。这里，在把代码提交给 assembler 之前，用 cpp 把两个头文件内嵌入汇编代码文件。Mips/asm.h 定义了宏 LEAF 和宏 END（见下面），mips/regdef.h 定义了惯用的寄存器的俗称（conventional name），比如 t0 和 a1(section 2.2.1)。

macro
这里我们用了 2 个宏定义：LEAF 和 END。它们在 mips/asm.h 中定义被如下：

```
#define        LEAF(name)  \
    .text;  \
    .globl  name;   \
    .ent        name;   \
name:
```

LEAF 被用来定义一个简单子函数(simple subroutine)，如果一个函数体内不调用其它函数，那幺相对于整个调用树(calling tree)而言，这个函数就是调用树上的一片“叶子”，因此得名“leaf”。相对的，一个需要调用其它函数的函数，叫“nonleaf”，nonleaf 函数必须多做很多麻烦的事情例如保存寄存器和返回地址，不过很少会真的需要自己写一个 nonleaf 的汇编代码（$：这通常用 C 语言来写)。注意下面：

.text　　表示这段用汇编写成的代码应该放在“.text”段中，“.text”是 C 语言程序的代码段。

.globl　声明“name”为全局变量，在模块的符号表(symbol table)中作为全局唯一的符号而存在($：全局变量在整个程序内唯一；局部变量在其所在函数体中唯一；static 变量在其所在文件内唯一)。

.ent　　对程序而言没有实际意义，只是告诉 assembler 将这一点标志为“name”函数的起始点，为调试提供信息。

.name　将其所在地址命名为“name”，作为 assmbler 的输出。名为“name”的函数调用将从该地址开始。

END 定义了两个 assembler 需要的信息，都不是必须的。

```
#define END(name)   \
    .size       name,   .-name; \
    .end        name
```

.size　　表示在 symbol table 中，“name”函数体的大小(字节数)将与“name”符号一道列出。

.end　　指出函数尾。调试用信息。

.set　　伪操作符(directive)，用来告诉 assembler 如何编译。

在本例中，.noreorder 表示禁止对代码重排序，让代码严格保持其书写的顺序，否则 MIPS assembler 会尝试将代码重新排序——填补那些 delay-slot 以获得较好的运行效率。Nowarn 要求 assembler 不要费心去指出那些应该被重排序的地方，相信程序员已经处理好这些事情了。通常这不是个好主意——除非你确信你肯定正确。基本上这是个不必要的 directive。Labels:“1:”是数字标志 label，大多数的 assembler 都会把它当作**局部**label 来处理。像“1:”这种 label，在程序里你想用多少都可以：你可以用“1f”引用 reference 下一个“1:”；用“1b”来引用前一个“1:”。这会很常用。

Instructions：一些指令的顺序会有出乎预料的问题，你必须注意。.set noreorder 这一 directive 使得 delay-slot 问题变得非常敏感而容易出问题，我们必须确保 load 的数据不会马上被下一条指令用到。比如说：

```
    bne     t0, t1,.t01
lbu     t2, -1(a0)
    ..............
.t01:
    j       ra
subu        v0, t0,t1
```

这里 lbu t2, -1(a0)一句中，用 t2 不能用 t0，因为要执行的下一条指令 subu v0, t0,t1 中要用到 t0。

好，已经看过了一个例子，让我们再看一些语法方面的东西。

9.2 语法概要 Syntax Overview

在附录 B 中你可以找到 MIPS 汇编器的语法列表，大多数的其它厂商的编译器也都遵循这个列表的规则。当然，可能少数的 directive 的具体含义会有少许的差别。如果你以前在类 unix(unix-like)的系统上用过 assembler，那这个列表你应该会很熟悉。

9.2.1 Layout, Delimiters, and Identifiers

首先你得熟悉 C 语言，如果你熟悉 C，那幺注意，汇编代码与 C 代码有一些区别。

汇编代码以行为分界，换行（end-of_line）表示一个指令或伪操作符 directive 的结束。

你也可以在一行里写多条指令或伪操作符，只要它们中间用“;”隔离开来。

以“#”开头的行是注释，assembler 将忽略它。但是**不要把“#”放在行的最左面**：这将激活 C 预处理器 cpp(C preprocessor)，有时候你可能会用到它。如果你确定你的代码会经过 C 预处理器的预处理,那幺你可以在你的汇编代码中使用 C 风格的注释方式:“/*…*/”，可以跨越多行，只要你乐意。

变量和 label 的名字(identifiers)可以随意——只要在 C 语言里合法就行，甚至可以包含“$”和“。”。

在代码中你可以使用 0~99 之间的数字作为 label，它会被视为临时性的符号，所以你可以在代码中重复使用同一个数字作为 label。在一个分支指令（branch instruction）中“1f”指向下一个“1:”，而“1b”指向前一个“1:”，这样就不用费心为那些随手而写的跳转和循环起名字了，省下这些名称可以去命名那些子程序、还有那些比较关键的跳转。

MIPS/SGI assembler 通过 C preprocessor 的宏定义来提供寄存器的俗称（conventional name）（$：zero,t0,~,ra），所以你必须用 C preprocessor 来对你的汇编代码进行预处理，为此需要在代码中包含 include 头文件 mips/regdef.h。虽然说规范的 assembler 通常可以识别这些寄存器的俗称，但是为了代码的通用性起见，还是不要把宝压在这上面为好。

assembler 的定位计数器指向正在编译的当前指令的地址，你可以在汇编代码中引用 assembler 的定位计数器的值。标识符“。”代表 assembler 当前的定位计数器的值。你甚至可以对它做有限的一些操作。在上下文中，label(或者其它什幺可复位位的符号 relocatable symbol)，将被替代为它的地址。

（$：类似于 arm 里 adds r0,pc,symbol address - (。+8)这样的操作。）

固定字符和字符串的定义方式与 C 相同。

9.3 指令规则 General Rules for Instructions

Mips assembler 允许一些指令的简略写法。有时候，你提供的操作数 operand 少于机器码所要求的，或者机器码要求使用寄存器而你却使用了常数，在某些情况下，assembler 也会允许这种写法，并自动进行调整。你将会发现，在真正的汇编代码中这种情况非常频繁。这一节我们将讨论这个问题。

9.3.1 寄存器间运算指令

Mips 的运算指令有 3 个操作数。算术 arithmetical 或逻辑 logical 指令有 2 个输入和一个输出，例如：Rd = rs + rt，被写成 addu rd, rs,rt。

这里的 3 个寄存器可以重复(例如 addu rd, rd,rd)。在 CISC-style 的 cpu（例如 intel386）指令中，只有 2 个操作数，Mips assembler 也支持这种风格的写法，目的寄存器 destination register 可以同时作为一个源操作数 source operand：例如：addu rd, rs，这与 addu rd，rd，rs 相同，assembler 将自动将它转换成后者。

Mips assembler 提供的指令集中有一些伪指令 unary operation，比如 Neg，not，这些伪指令实际上是一条或多条机器指令的组合。对这些指令，Assembler 最大接受 2 个操作数。Negu rd, rs 实际上被转化为 subu rd, zero,rs，而 not rd 将被转化为 or rd，zero，rs。

可能最常用的寄存器间操作 register-register operation 要算是 move rd，rs 了。这条指令实际上是 or rd，zero，rs。

9.3.2: 带立即数的运算指令

在 assembler 和机器语言里，嵌入在指令中的常数被称为立即数 immediate value。很多 Mips 的算术和逻辑指令都有另外一种形式，这种形式里 rt 寄存器被一个 16bit 的立即数所取代。在 cpu 的内部运算过程中，这个立即数将被扩展为 32bit，可能是符号扩展 sign-extend（$：用最左面的 bit(bit15)填充扩展的高 16bit），也可能是零扩展 zero-extend（$：用 0 填充扩展的高 16bit)——这取决于具体的指令。一般而言，算术指令进行符号扩展 sign-extend，而逻辑指令进行零扩展 zero-extend。

在机器指令的概念上，即便执行同一种运算，操作数中是否包含立即数的区别，将导致两条不同的指令（例如 add 与 addi）。尽管如此，对于程序员而言，还是没有太大的必要去具体的区分那些包含立即数的指令。Assembler 会找出它们，并进行转换。比如：

addu $2，$4，64 ————————> addiu $2，$4，64

如果立即数过大而超过了 16bit 所能表达的范围，机器码中将无法容纳，这时 assembler 会再次帮助我们：它会自动将立即数载入“编译用临时寄存器 assembler temporary register”at/$1 中，然后进行如下操作：

add $4， 0x12345 ————————> li at，0x12345
add $4，$4，at

注意这里的“li”(load immediate)指令，在 cpu 提供的机器指令集中你找不到它。这是一个及其常用的宏指令，用来把 32bit 整数装载入寄存器，而不用程序员来操心怎么去实现这一动作：

当这个 32bit 整数值介于-32k~+32k 之间，assembler 用 addiu 指令配合 zero 寄存器来实现“li”;

当 16-31bit 为 0 时，用 ori 指令来实现“li”;

当 0-15bit 为 0 时，用 lui 指令来实现“li”;（$：运算指令（ori、addiu）要比存取指令（sw、lui）的处理速度快）

如果以上条件都不成立，那只好用 lui/ori 两条指令来实现“li”了：

```
li   $3，-5          ————————>      addiu   $3，$0，-5
li   $4，08000       ————————>      ori     $4，$0，08000
li   $5，120000      ————————>      lui     $5，0x12
li   $6，0x12345     ————————>      lui     $6，0x1
ori      $6，$6，0x2345
```

9.3.3　关于 32/64 位指令

我们在前面（2.7.3）讲过可以对 32 位指令的机器码进行符号扩展到 64 位，以保证 32 位的程序（mipsII）在老的机器上能正常运行。

9.4 地址模式

前面提到过，mips cpu 硬件上只支持一种地址模式：寄存器基地址+立即数偏移量 base_reg+offset,偏移量必须在-32768 到+32767 之间(16bit 带符号整型所能表示的范围)。但是 assembler 可以通过一些方式来支持以下几种地址模式：

Direct：由你提供的数据标号或外部变量名。

Direct+index：一个偏移量，加上由寄存器指出的标号(label)地址。

Constant：一个数字，作为一个 32 位的绝对地址(absolute address) 处理。

Register indirect：是寄存器加偏移量的特殊形式：偏移量为 0。

9.5 assembler directives

MIPS 中所有指令都被塞在 32bit 空间里的做法导致了一个明显的问题:访问一个确定的/嵌入在指令以内的?????(compiled-in location)内存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变量的程序中,这一缺陷往往导致最后编译出的代码臃肿而低效.

早期的 MIPS 编译器引入了一种技术以弥补以上缺陷,这项技术被以后的 MIPS 编译工具链 toolchain 一直沿用下来,它通常被称为”全局量指针相对寻址”gp-relative address.这个技术要求 compiler,assembler,linker 以及运行时激活代码(runtime startup code)偕同配合,把程序中的’小’变量和常数汇集到一个独立的内存区间;然后设置 register $28(通常称为全局量指针 global pointer 或简写为 gp)指向该区间的中央(linker 生成一个特殊符号_gp,其地址为该区间的中央,激活代码负责将_gp 的地址加载到 gp 寄存器,这一动作在第一个 load/store 指令运行之前完成).只要这些全局变量\静态编量\常量加起来不占用超过 64k 大小的空间,这些资料相对该区间的中点也就不超过正负 32k(偏移量 15bit+符号位 1bit,参见 mips 机器码格式),那幺我们就可以在一条指令中完成对它们的 load/store 操作:

```
lw  $2, addr        ------>    lw  $2, addr-_gp(at)
```

一个问题是在编译彼此独立的模块的时候,compiler 和 assembler 如何决定哪些变量需要通过 gp 来寻址,通常的做法把所有小于某个特定长度(通常是 8byte)的对象放进该区间,这个长度可以通过 compiler/assembler 的”-G n”选项来控制.特别需要指出: ”-G 0”将取消 gp-relative 寻址方式.

上面所说的 gp-relative 寻址是一种非常有用的技巧,然而在使用中会有一些”陷阱”值得注意.在汇编代码中声明全局量的时侯你最好小心点:
可写,且初始化过的小对象体 writable,initialized small data 必须显式的声明在.sdata 段中.(“小对象体”一词中“小”的含意即为上面提到的”长度小于 8byte”)
声明全局对象时必须指出其长度.

```
.comm.  smallobj,   4
.comm.  bigobj, 100
```

声明小外部变量时同样需要指出其长度.

```
Extern  smallext,   4
```

大多数 assembler 不会对对象声明作辅助性的处理(如指出该对象的长度).
C 代码中的全局变量,必须在所有使用它的模块中被声明.对于外部队列,你可以显式的指出它的长度,如:

```
Int cmnarray[NARRY];
```

也可以不指出其长度:

```
extern  int exarray[];
```

有时候,程序运行的方式(环境)决定了不能采用 gp-relative 寻址方式.比如一些实时操作系统,还有很多固化环境下的程序(PROM monitor)是用一块单独连接 link 的代码实现来内核 kernel,应用程序直接使用子函数(而不是通常的系统调用)调用到内核中去.这种情况下无法找到一个和适的方法以使 gp 寄存器在内核和应用程序的两个小数据段.sdata 中来回切换,所以内核与应用程序两者之一(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”选项编译模块的时候,那些需要与该模块连接的库 library 通常也应该使用”-G 0”选项编译.在资料是否应该放在.sdata 的问题上, 模块和库的声明应该彼此一致,如果发生冲突的话,linker 将无法判定资料应该放在小数据段还是普通数据段,这时 linker 会给出奇怪而毫无价值的错误信息.

9.5 Assembler Directives

在一开始我们就已经提到过”directive”,你也可以在附禄 B 里找到它的清单,不过没有详细介绍.

9.5.1 段的选择

通常的数据段和代码段的名字以及对它们的支持在不同编译工具链上可能会不一样.但愿大部分至少能够支持一般的 MIPS 通用的段,见图 9.1.
在汇编代码中,以如下方式来选择段:

```
.text, .rdata, and .data
```

简单的把适当的段名放在数据和指令之前,就象下面的样子:

```
.rdata
msg:    asciiz  “hello world!\n”

    .data
table:
    .word   1
.word   2
.word   3

    .text
```

```
func:sub    sp, 64
.................
```

.lit4 and .lit8 : 隐式浮点常数段 floating-point implict constants
你不能像 directives 一样写这些段.它们是由 assembler 隐式创建的只读数据段,用来放置 li.s 和 li.d 宏指令中的浮点常数型参数.一些 assembler 和 linker 会合并相同的常数以节省空间.
.bss, .comm., and .lcomm data
这个段名也不用作 directive,它用来收集在 C 代码中声明的所有未初始化的资料.C 的一个特点是: 在不同的模块中可以有同名的定义.只要其中被初始化的不要超过一个..bss 段用来收集那些在所有模块中都没有初始化过的数据.fortran 程序员可以认为这个就是 fortran 语言中的.common 段,虽然名字不一样.
你必须声明每个资料的长度(单位为 byte),当程序被连结的时候,它就可以得到足够的空间(所有声明中的最大值).如果有任何模块把它声明在初始化过的数据段,这些长度将被用到,并且使用如下声明:

```
.comm.  dbgflag,    4       #global common variable, 4 bytes
.lcomm. sum,    4       #local common variable, 8 bytes
.lcomm. array,  100     #local common variable, 100 bytes
```

“未初始化 uninitialized”这一说法实际上并不准确:虽然这些段在编译出的目标文件中是不占地方的,但是在执行你的程序之前,运行时激活代码 run-time startup code 或操作系统会将.bss 段清零----------很多 C 程序都依赖于这一特性.
.sdata, small data, and .sbss
这些段被编译工具链用作单独放置小资料对象的.data 和.bss 段.MIPS 编译工具链进行这个处理是因为,对一个足够紧凑的小资料对象段可以进行高效率的 load/store 操作,其原理是在 gp 寄存器中保存一个资料指针,具体说明见本书 9.4.1 章节.
注意,.sbss 不是一个合法的 directive;放在.sbss 段中的资料满足两个条件:1,用.comm 或.lcomm 声明;2,其长度小于”G n”编译选项所指定的长度(默认为 8byte).
.section
开始一个任意名称的段,并提供旗标(可能在代码中提供,也可能是工具包提供? Which are object code specify and probably toolkit specific).查看你的工具包的说明手册,??????????????????????
如图 9.1 所示的结构可能适合做为一个运行在裸机 bare cpu 上的固化程序 ROM program.只读段倾向于放在远离下部可擦写区间的内存位置上.
堆 heap 和栈 stack 并不是真正的能被 assembler 或 linker 所识别的段.一般的,它们在运行时由运行系统???????run-time system 初始化和维持.通过把 sp 寄存器设置为该程序可用内存的最高地址(8byte 对齐)的方式来定义栈;堆通过一个由类似于 malloc 函数使用的全局指针变量来定义,通常初始化为 **end** 符号 symbol,该 symbol 被 linker 赋值为已声明变量的最高位置.

特殊符号
图9.1 显示了一些由 linker 自动声明的符号,以便程序找到自己各个段的起始\结束的位置.这最初只是习惯,后来在 unix 类的系统上得到发展,其中一些是 MIPS 环境中所独有的.你的工具包手册上可能定义了他们中的一部分或全部;下面打@的表示肯定会被定义的符号:

symbol	standard	value

_ftext　　代码段起始点
etext　　@　　代码段结束点
_fdata　　数据段起始点
edata　　@　　数据段结束点
_fbss　　未初始化段起始点
end　　@　　未初始化段结束点
(end 通常也就是程序 image 的结束点)

9.5.3　资料的定义与对齐

选择好正确的段之后,现在你需要用下面所说的 directive 来定义资料对象本身.

.byte, .half, .word, and .dword

这些 directive 产生 1,2,4,8byte 长度的整数(有些工具链-----即便是 64 位的-------没有提供.dword directive).可以跟随着一个值的列表,彼此以逗号分离,可以在值的后面加冒号并跟随一个重复计数,以表示连续几个相同的值,如下(word=4byte):

```
    .byte   3           #1 byte:            3
    .half   1,2,3       #3 halfwords:   1   2   3
    .byte   3           #5 words:       5   5   5   6   7
```

注意数据的位置(相对于段的起始处)在资料被输出之前自动对齐到合适的边界.如果你确实需要输出未对齐的资料,那幺必须自己使用西面要讲到的.align directive 来说明.

.float 和.double

这些 directive 输出单精度\双精度的浮点值.如下:

```
.float      1.4142175           #1 个单精度浮点数
.double     1e+10,  3.1415      #1 个双精度浮点数
```

与对整数处理相同,可以使用冒号表示重复.

.ascii 和.asciiz

这些 directive 输出 ASCII 字符串, 附带\不附带结束标记.如下两行代码输出相同的字符串:

```
.ascii      “Hello\0”
.asciiz     “Hello”
```

.align

这个 directive 允许你为下一个资料指定一个大于正常要求的对齐边界??????alignment.该 alignment 表示为 2 的 n 次方.

```
.align  4       #对齐到 16byte 边界 (2^4)
var:
.word   0
```

如果标志(上例中的 var)后面紧跟着.align,那幺这个标志仍然可以被正确的对齐,例如下面的例子与上面的例子作用相同:

```
var:
    .align  4   # 对齐到 16byte 边界(2^4)
    .word   0
```

对要求紧凑结构????packed 的数据结构,这个 directive 允许你取消.half,.word 的自动对齐功能,你可以指定它为 0 对齐,它将持续作用直到下个段开始.??????

```
.half   3       #正确对齐的半字
.align  0       #关掉自动对齐功能
.word   100     #按照半字对齐的字.
```

.comm 和.lcomm
通过指定对象名和长度来声明一个 common 或者说未初始化的资料对象.
用.comm 声明的对象对所有声明过它的模块有效,它的空间由 linker 分配,采用所有声明中的最大值.但是,如果其中有任何一个模块将其声明在.data,.sdata,.rdata,那幺所有的长度声明都将失效,而采用初始化定义取而代之.
??????.comm 的用途是为了避免以下情况:一个资料对象要在许多文件中用到,但它与其中的每一个文件都没有更特殊的联系,但是我们不得不在某个文件中声明它,这样就造成了不对称.但是它确实存在,因为 fortran 就是用这样的语意来定义它的,而我们想要经过汇编语言来编译 fortran 程序(比如查看 fortran 程序编译出来的汇编代码).
用.lcomm 声明的对象是局部对象,由 assembler 在.bss 段或.sbss 段为其分配空间,但是在所属模块之外它不可见.
.space directive 增加当前段的空间计数,例如:

```
struc:  .word   3
    .space  120     #空出 120byte 大小的空间
.word   -1
```

对通常的资料\代码段而言,这个空出的空间用 0 填充,如果 assembler 允许你声明内容不在对象文件中定义的段(如.bss),这个空间只影响连续的符号\变量之间的偏移.
9.5.4 符号绑定属性 symbol-binding attributes
符号 symbol(在数据段或代码段中的标志)可以被调节为可见?????,并可以供 linker 使用以便将几个分离的模块编译成一个完整的程序.
符号有三个级别的可见度:
局部:
除了声明它的那个模块,它对外部而言是不可见的,并且不会被 linker 使用.你不用担心在其它模块中是否适用了相同的符号.
全局:
这些是公开的符号以供 linker 使用.使用.extern 关键词,你可以在其它模块中引用全局符号,而不必为它定义本地空间.
弱全局 weak global:
这个晦涩的概念在一些工具链中以关键词.weakext 实现.它允许你定义一个 symbol,如果有同名的全局对象存在,那幺就把它连结到这个同名的全局对象;如果不存在同名的全局对象,那幺它就作为一个局部对象存在.如果.comm 段存在,你就不应该用’弱全局’这个概念.????
.globl
在 C 语言环境下,除非用 static 关键词进行声明,否则模块级的数据和函数入口都默认为全局属性.与 C 语言不同,一般在汇编语言环境下,除非使用.globl directive 显式的进行声明,否则标志 label 默认为局部属性.对于用.comm 声明的对象不需要再使用.globl,因为它们已经自动具备全局属性.

```
    .data
    .globl  status      #全局变量
status: .word   0

    .text
    .globl  set_status      #全局函数入口
set_status:
```

```
    subu    sp, 24
……..
```

.extern

如果引用当前模块中未定义的标志,那幺(assembler)将假定它是在”其它模块中定义的全局对象”(外部变量).在一些情况下,如果 assembler 能知道所引用的对象的长度,它就可以生成更优化的代码(见 9.4.1 节).外部变量的长度用.extern directive 来指明:

```
.extern index,  4
.extern array,  100
lw      $3,index        #提取一个 4-byte(1-word)长度的外部变量
lw      $2,array($3)        #提取 100-byte 长度外部变量的一部分
sw      $2,value            #装载一个未知长度的外部变量
```

.weakext

一些assembler和工具链支持**弱全局**的概念,这允许你为一个符号 symbol 指定一个暂时性的绑定(binding,是一个连接用的概念,指符号与其内存地址间的对应关系?????),如果存在一个正常全局(强全局)对象定义,那幺它将把先前的这个**弱全局**绑定覆盖.例如:

```
.data
.weakext    errno
errno:  .word   0

.text
    lw  $2, errno       #可能使用局部定义,也可能使用外部定义.
```

如果没有其它模块使用.globl 来定义 errno,那幺这个模块------还有其它模块------将会使用上面代码中 errno 的局部定义.

另外一个可能的用法是:用一个名字声明一个局部变量,而用另一个名字声明它另外的**弱全局**身分.

```
    .data
myerrno:    .word   0
    .weakext    errno,  myerrno

    .text
    lw  $2, myerrno         #总是使用上面的局部定义
    lw  $2, errno           #可能使用局部定义,也可能使用其它的
#(外部定义)
```

9.5.5　函数 directive

附录 A 指令的时序和优化

MIPS CPU 高度流水化，所以它们执行代码的速度依赖于流水线的工作情况。有某些情况下，代码的正确性依赖于流水线的工作方式---特别是使用CPU控制协处理器0的指令和寄存器。通过显式使用寄存器而传递的依赖性相当明显，只不过比较凌乱。除此以外，在隐式使用的寄存器中也有一些偶然的依赖关系。例如状态寄存器中的 CPU 控制标志会影响到所有指令的指令的执行，改变这些标志必须非常小心。

大部分 MIPS 指令需要在流水线 RD 阶段的结束时得到它们的操作数，并且要在随后的 ALU 阶段的结束时产生这些指令的运行结果，这如图 A.1 所示的那样。如果所有的指令总能够遵守这些规则，任何指令序列都能够以最大速度正确的运行。在 MIPS 架构中最大的奥妙就是绝大多数指令都能遵守这些规则。

在由于某些原因而不能做到的情况下，使用从前面的紧邻指令处得到操作数的指令不能及时正确的运行。这种情况能被硬件检测到，然后通过延迟第二条指令直到数据准备好以使之得到修正，或者它可以留给程序员来避免产生试图使用未准备好的数据（pipeline hazard 流水线冒险）的指令序列。

A.1 避免冒险：使代码正确

可能的冒险包括下面这些情况：
1．load 延迟：在早期的 MIPS CPU 中的这是一种流水线冒险；紧跟着 load 指令后面的指令不能引用由 load 产生的数据。有时候当没有有用的东西能被安全的移到被延迟的指令槽时，需要编译器/汇编器使用一条 nop 指令。但从 R4000 开始，MIPS CPU 已经是互锁的了，这样冒险就不会影响到通常的用户级指令。
2．乘法单元冒险：从 MIPS CPU 的整数乘法器得到的运算结果是互锁的，所以取得这个运算结果的 mflo 指令没有延迟指令槽。但是整数乘法硬件的独立性产生了自己的问题，参看 A.3 节。

3．协处理器 0 冒险：协处理器 0 控制指令通常用不同于平常的时序来读/写寄存器，这样就产生了流水线问题。其中多数没有互锁。详细信息必须从你的 CPU 用户手册中查，但是我们将看一下你在 R4000 CPU(可能是 MIPS CPU 中最难处理的)上必须要做的事情。

注意分枝延迟指令槽，尽管它是为了降低流水化而被引入的，但它作为 MIPS 架构的一部分，因此不再是冒险了；它只是一个特例而已。

A.2 避免互锁来提高性能

只要 CPU 发生互锁，我们将会损失性能。但是如果用一些巧妙的法子，CPU 本来是可以做些有用的事情的。我们想让编译器（or for heavily used function perhaps a dedicated human programmer）重新组织代码来得到最佳运行效果。

编译器—以及人—都发现这是一个挑战。一个高度优化已避免互锁的程序经常将运算的几个阶段分解，然后交错的执行，这样就很难看出将会发生什么。如果代码只是从原来的位置被前后移动了四，五条指令，通常还好处理。更大的移动就会出现越来越大的问题。
在单流水线机器（目前的绝大部分 MIPS CPU）中，绝大部分指令使用一个时钟周期，所以对那些用四到五个时钟周期才能完成的指令以及那些成功的和其他指令交迭的指令，我们都有希望重新组织它们。在 MIPS CPU 中，这些标准只有对那些浮点指令才能很好的符合，所以高级调度机制会提高浮点指令的性能但对整数指令却作用无几(1)。深入讨论这个问题超出了本书的范围；如果你想很好的回顾一下所使用的编译器技术，请看一下 Hennessy and Patterson, <<计算机体系结构：一种量化的方法>>。如果想知道各种 CPU 的详细时序，请查一下相应的用户手册。
在第十二章第 388 页有一个小规模的关于 load 互锁的代码优化的例子。

A.3 乘法单元冒险：早期修改 lo and hi。

当一个 MIPS CPU 发生了中断或异常时，大部分流水线中的指令被中止，并且禁止将运算的

结果写回。但是整数乘法单元很少与 CPU 的其余部分有关联，因此继续运行， 这并不会对异常产生影响。这意味着一旦乘法和除法指令开始以后，不能防止改变乘法单元的运算结果寄存器的 lo and hi。
异常可能及时发生以防止 mfhi 或者 mflo 完成写回操作，但可能允许后续的乘法或者除法指令开始运行—并且一旦第二个运算开始以后原来的数据将会丢失。
为了避免这个问题，确保至少用两个时钟周期从后面的乘法或除法指令分离 mfhi 或者 mflo 指令，那么在所有的 MIPS CPU 上都是足够的了。好的编译器和汇编器将会为你处理这些的。而且只有你反汇编这些代码，你才会知道它的存在，这样你会发现一些没有料想到的 nop 指令。

---

1. 这是为何 SGI 编译器在高度浮点化程序上快得多的一个原因—可能快 30%，但在整数代码上比 GNU C 差一点点。

A.4 避免协处理器 0 冒险：有多少 nop 呢？
程序员的问题是，我需要在一对特定指令之间放多少条指令（很可能是 nops）才能让它们安全的工作呢？
原则上是可以得到指令对和在它们之间需要多少时钟周期的一份完整清单。但是那太费时间了。但我们能够降低这个工作的规模，我们注意到只有在以下情况时问题只会出现：

1．使用比标准时间（标准时间就是 ALU 阶段的末端）长的时间来产生数据的指令和/或
2．指令需要使用在标准时间（在这种情况下，标准时间是 ALU 流水阶段的开始）前准备好的数据

我们不需要列出在标准时刻产生和使用数据的指令，只要列出那些偏离正常途径的指令就可以了。对于其中的每一条指令，当运算结果产生了以及/或者当需要操作数时（1）我们都需要当心。有了这些，我们将能够在最复杂的情况下得到正确的或者高效的指令序列。
表 A.1 展示了 R4000/4400 CPU 的时序，这张图原来出现在 Heinrich,<<R4000/R4400 用户手册>>(在参考书目可以找到 Web 地址)中。这个表列出了操作数被使用和运算结果对后续的指令变为可用的流水阶段。

1. 列出运算结果迟了多少时钟周期的或者操作数早了多少时钟周期将是足够的---也是最简单的。但是 MIPS 系列的图表采用了流水线阶段。

表 A.1 有可能冒险的协处理器 0 指令和 R4000/R4400 CPU 的事件时序

MIPS 的演化

MIPS16 是一个 1997 年面世的可选的指令集扩展，它能减少二进制程序尺寸的 30-40%。实现

者希望这种 CPU 能够在很关心代码尺寸的场合中更有吸引力--这种场合通常就是指低成本系统。由于只应用于特定实现，它是一个多厂商标准：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 二进制代码比其他架构的并不是 MIPS 指令集干的活少了，而是他们的尺寸更大一些--每个指令 4 字节长，相比之下某些 CISC 架构一般平均只有 3 个字节。

MIPS 增加了一种模式，在这种模式下 CPU 可以对 16 位固定大小的指令进行解码。大多数 MIPS16 指令扩展成正常的 MIPS III 指令，所以很明显这将是一个相当受限制的指令子集。窍门就在于使这个子集对足够多的程序充分的进行高效编码，以使整个程序的大小得到大大的压缩。

当然，16 位指令并不会使其变成一个 16 位指令集。MIPS16 CPU 是实际存在的带有 32 位或者 64 位寄存器的的 CPU，MIPS16 CPU 的运算也都在这些寄存器上。

MIPS16 远不是一个完整的指令集—例如它既没有 CPU 控制指令，也没有浮点运算指令。但没有关系，因为每一个 MIPS16 CPU 也必须要运行完整的 MIPS ISA。你能运行 MIPS16 和正常的 MIPS 代码的混合指令。每个函数调用或者跳转-寄存器指令都能改变运行模式。

1. 并不是 MIPS 发明了提供一种可选的使部分指令只有一半大小的创意。Advanced RISC Machine（ARM）公司的 Thumb 版本的 ARM CPU 首先提出这个想法的。

在 MIPS16 中把指令地址编码成最低有效位(Least Significant Bit, LSB)模式是既方便又高效的。MIPS16 指令必须偶字节对齐，所以 bit 0 不再是指令指针(instruction pointer, 就是程序计数器 PC)的组成部分了；取而代之的是，每条跳到奇数地址的指令开始执行 MIPS16，每条跳到偶数地址的指令回到正常的 MIPS。MIPS 子程序调用指令 jal 的目标地址总是字对齐的，所以新指令 jalx 隐藏了指令的模式间转换。

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 bit 来选择寄存器，这样只有 8 个通用寄存器允许自由访问；在许多 MIPS 指令中可以见到的 16 bit 常数域也被压缩，通常变成了 5 bit。许多 MIPS16 指令只指明两个寄存器，而不是三个。另外，还有一些特别的编码规则将在下一节描述。

D.1.1 MIPS16 中的特殊编码格式和指令

被缩减的通用指令没有什么问题，但有两个特定的弱点会加大程序尺寸；5 bit 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地址范围。三种新的指令和一种特别规定有助于解决这些问题。

extend 是一条特殊的 MIPS16 指令，它由 5 bit 的代码和 11 bit 的域构成。这个 11 bit 的域可以和后续指令中的立即数域相连接，这样就允许使用一个指令对来对 16 bit 立即数编码。这条指令在汇编语言中看起来就像一个指令前缀。

装载(load)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 对相对于指令自身位置的装载操作(PC-relative loads， PC 相关装载)增加了支持，允许常量被嵌到代码段中(典型情况就是在函数的起始处前面)。这些是仅有的不是严格对应于正常的 MIPS 指令的 MIPS16 指令—MIPS 没有 PC 相关的数据操作。

许多 MIPS load/store 操作是直接在栈帧(stack frame)里，$29/mp 可能是最普通的基寄存器。MIPS16 定义了一组隐式使用 mp 的指令，允许我们把函数的栈帧引用地址也编进去而不需要一个分离的寄存器域。

MIPS 的 Load 指令总是生成 32 位的全地址。由于装载字(load word)指令只有当地址是 4 的倍数是才合法，最低两位就被浪费了。MIPS16 的 Load 指令是可以伸缩的：地址的偏移量会根据被 load/store 的对象的大小左移，这样就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 可访问的的寄存器中的一个与 32 个 MIPS 通用寄存器中的任何一个间任意做数据移动。

D.1.2 对 MIPS16 的评价

MIPS16 对于汇编语言编程来说不是一种合适的语言，我们也不准备对它详细说明。这些是编译器的工作。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%。MIPS16 比 32 位 CISC 架构的代码更紧凑，和 ARM 的 Thumb 代码差不多，和纯 16 位 CPU 相比相当有竞争力。
但是没有免费的午餐；MIPS16 程序可能比 MIPS 增加 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 经常主要被存储器所限制，而不是被 CPU 核所限制。较小的 MIPS16 程序需要较低的带宽来取指令，这样就得到更低的 cache 缺失率。在 cache 很小并且程序的存储器有限时，MIPS16 将会弥补差距，还有可能要重新改写正常的 MIPS 代码。
由于性能的降低，MIPS16 代码在有大的存储器资源和很宽总线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。
在应用范围的另一端，MIPS16 将会与软件压缩技术展开竞争。在放进 ROM 存储器之后，使用通常的文件压缩算法压缩的正常 MIPS 程序将会比未压缩的同等 MIPS16 代码小，而稍大于压缩过的 MIPS16 同等代码(注 1)；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，那么全 ISA 软件解压很可能会带来更好的总体性能。也有这样一种趋势来构造系统，那就是大量使用以字节编码的解释语言(Java 或者它的后续者)来书写大量在时间上要求不严格的程序。那种中间代码非常小，在尺寸方面比任何二进制机器码都高效的多。如果只有解释器和一些对性能要求严格的程序留在机器中 ISA 中，那么更密集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)本身会非常大，但是应用复杂度的无情增长将很快使它减少重要性。
我预料在 1998-2003 年将会看到 MIPS16 小范围的应用于低能量、小尺寸和成本受限制的系统中。它还是值得发明的，因为有些系统—比如”智能”移动电话—可能会大量生产。

1．更密集的编码格式在使用上比压缩算法有更低的冗余度。

D.2 MIPSV/MDMX
MIPS V 和 MDMX 是在 1997 年早些时候一起公布的。它们本来是为一种新的准备在 1998 年发布 MIPS/SGI 的 CPU 中的指令而设计的。但是那个 CPU 后来被取消了，关于它们的未来存在疑问。
二者都是为了克服一些已知的传统指令集的不足，这些不足是在 ISA 面向多媒体应用中产生的。象软调制解调器的语音编/解码、或流媒体应用、或图像/视频的压缩/解压缩这样的任务采用一些过去只有专用数字信号处理器(digital signal processor, DSP)才用的数学算

法。在这种计算等级，多媒体任务通常都包括重复进行一些对大向量或者数组数据的相同操作。

在基于寄存器的机器内部，通常采用的方案是把多媒体数据项封装到一个机器寄存器中，然后执行一条寄存器-寄存器指令，这条指令对于每个寄存器中的每个域做同样的工作。这是一种非常明显的并行处理形式，被称为单指令，多数据(single instruction, multiple data. SIMD)。

这个想法首先见于一款 Intel 的业已消失的 i860 架构的微处理器(circa 88)中。作为对 Intel x86 指令集进行扩展的 MMX 在 1996 年投放市场后，SIMD 重新登场时更加引人注目。MDMX 对操纵在一个 64 位寄存器中 8x8-bit 的整数组提供了一组操作，这些操作能够对所有的 8 小片做同样的事情。这些指令包括通常的算术操作(加，减，乘)，也有乘法-累加指令能把结果放在一个巨大的累加器中，这个累加器有足够的精度防止溢出。

由于这些指令被用于特定数据类型被相当清楚的从正常的程序变量分离开来的场合中，MDMX 指令集与浮点寄存器一起工作就变得有意义。以这种方式重复利用现有的寄存器意味着现有的操作系统不需要改变(在任务切换时操作系统已经保存和恢复浮点寄存器了)。

与 MDMX 相似，Intel 的 MMX 为封装进一个 64bit 的 8 个 8bit 数提供了”octibyte”八路(eight-way)指令。MIPS MDMX 也定义了 4x16 位(四个短整数操作)和 2x32 位(两个字操作)格式，但是早期的情况是一些 MDMX 实现可能认定 octibyte 格式和指令足够了。

当对 8bit 的数做算术运算时，结果经常下溢和上溢。如果我们必须为众多的溢出测试条件编写处理程序，那么多媒体应用的性能将不会得到提高。而只简单截去最大的和最小的数(对于无符号 8-bit 数来说，就是 255 和 0)的上溢和下溢结果，对于机器运算来说会更加有帮助。这个处理过程叫做”饱和”(saturating)算法。MDMX 拥有这种能力。

这就给我们带来了 MIPS V。尽管从名字上看好像意思是指一个升级的指令集--就像 MIPS I 到 IV 那样，MIPS V 在浮点领域跟 MDMX 很相似，提供了 paired-single 操作。paired-single 对一对被封装进 64-bit 的浮点寄存器中的单精度数做两次 FP 动作。

MIPS V 没有 MDMX 那么古怪；MIPS IV 包含了一个相当广泛的浮点运算集合，并且直接为其中的绝大部分提供了 paired-single 版本的指令；甚至成对比较(paired-compare)也可以做到，这是因为 MIPS IV 的 CPU 已经有了多个浮点条件位来接收结果。但 MIPS V 没有提供复杂多周期指令的成对操作版本的指令，这些多周期指令会需要非常多新的资源(例如没有求平方根和除法)。

D.2.1 编译器能用多媒体指令吗？

引入SIMD多媒体指令的原因和70年代晚期以前在超级计算机中提供向量处理单元的原因相似。很容易为向量处理器构造一个手工矩阵算术包。而用向量运算来编译一个用高级语言写成的程序就难得多了，尽管超级计算机提供商在这上面也取得一些成果。通常这些成果都集中在 Fortran 上；对于常规编程来说语义上的弱点使 Fortran 成为一种可怜的语言，但是这让它变成了一种很容易优化的语言，因为边际效应非常明显。

人们一致认为向量化的Fortran编译器在旧的程序上工作的不是很好(“dusty decks”，一句迷人的 Fortran 行话)。这样的编译器要求编程人员书写或者修改程序中的循环来使适应优化器的要求，这样才能带来显著的好处。这样可能是一种好的分工：循环可以使用固定格式，但程序员还可以将它们理解为顺序代码，但实际上这些循环编译的结果是一些很难懂的并行代码。术语”optimizer friendly”是含糊不清的：并行处理理论将会把它定义为：“特

定种类的边际效应的缺少，尽管实际的编译器可以查找遵循一些严格的多的规定的循环，这样哑模式匹配器就可以将它们安全的识别出来，然后进行向量化处理”。
C 的向量化困难的多。这是因为它使用的内存和基于指针的模型，这种模型对于任何数组访问都是隐式操作的。这使得除了最简单的循环以外很难消除其他的边际作用。在产品化上还没有做多少工作。
由于这段历史，开发能够采用多媒体 SIMD 指令成功优化程序的 C/C++编译器的前景如何呢？我猜测在最近前景不好。Intel 的 MMX 是最广泛使用的现代 SIMD 指令集，但当前也只有汇编语言用户才会使用（注 1）。我不希望看到使用 x86 MMX 的编译器。如果 MMX 得到大规模的成功使用，并且依赖于汇编子程序，结果就会是这些程序被 x86 架构所束缚住了；this would hardly be something that Intel would be in a hurry to change.
很多人预测在 1998 年或者 1999 年 Intel 会引入一种更好的 ISA 扩展。这种扩展将会增加更多的数据格式到”MMX 的后代”中，包括成对的单精度浮点数。如果这种能力更强的指令得到编译器的支持，那就可能出现很多同时适用于 MIPS V 的软件。

D.2.2 使用 MDMX 的应用程序

就像 x86 MMX, MDMX 对 3D 图像和视频应用将会比较有用，在这些应用中 CPU 把像素值推给软调制解调器所需的低精度信号处理单元。
不幸的是，”near display”3D 渲染的性能依赖于谨慎的与显存的集成。 甚至配置很好的 CPU 也竞争不过廉价的 PC 世界的加速器，这些加速器与大显存之间无缝结合(注 2)。图像和视频处理应用确实在这个等级上运行访问像素的程序，尽管这些这都是些桌面 PC 应用。
软调制解调器可能对那些希望使用电话的低端消费品设备会比较有用。它们和便宜的集成式调制解调器设备展开竞争，在更大范围里它们要与不断发展的进入家庭的数字电话竞争。
在我看来 MDMX 在游戏控制台上与 CPU/视频系统紧密结合将会是最好的机会。

D.2.3 MIPS V 的应用
成对单精度浮点指令和格式是为在高端图像和多媒体应用中出现的重复浮点计算增加带宽。尽管看起来象是硅图像公司(SGI)的市场，3D 图像使用的增长会使这种能力在更广的范围里更有有用。
对于 MIPS 有限的编译器支持看起来比对 MDMX 的支持更为真实。尽管成对操作看起来好像是超标量 CPU 的双发射指令的一种替代，它们实际上是相互补充的。SIMD 指令采用的并行机制来自于编译器中的更高一级操作，而低级调度还可以可能同时发射两条指令：一条对式浮点指令，另外一条指令负责整数或者管理操作。

1．一个愤世嫉俗的人可能会说由于任何 x86 的克隆都需要 MMX，那么 MMX 够用了。他还会说 MMX 到底是不是真的在用的问题已经离题太远了。而且从中受益的游戏和图像程序都是那些疯狂的汇编爱好者编写的。
2．如果有人使用与大内存和集成式视频刷新数据通道无缝结合的方式构造 CPU 的话，它们还有可能展开竞争。但是我没有看到那种 MIPS 产品的更多迹象。

D.2.4 MDMX/MIPS V 有可能成功

SGI 1997 年做出的放弃发展它的 H1 高端处理器项目的决定使这两种指令集(译者注：MDMX 和 MIPS V)一直没能正式发布。但是我相信至少有一种面向嵌入式市场的 CPU 会支持 MDMX。到底会发生什么是很有趣的。

在目前为止没有 CPU 支持 MIPS V；但是它比 MDMX 有更长的生存周期，将还会是 1999 年发布的 CPU 有用的附属物。

See MIPS Run 第三章

翻译：张福新
系统结构实验室
中国科学院计算技术研究所
2003 年 8 月 8 日

第三章

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,提供可选项配置方法以及某种观察或控制诸如高速缓存 (cache) 和时钟等片上功能的途径。但要用一个干净的、和具体实现无关的方法来描述这些东西很难,不象指令集中表示运算功能那么简单。

为了更便于读者理解，我们会把不同的功能分成几章来介绍。这一章里我们先介绍用来实现这些特色功能的公共机制。在读后续的三章之前，您应该先读本章的前面部分，特别要注意“协处理器”(下面将有解释)一词的含义。

那么， MIPS CPU 的协处理器 0 (以下简称 CP0)做些什么工作呢？

配置： MIPS 硬件常常是很灵活的，您可能可以选择一些很根本的 CPU 特性(例如大尾端/小尾端，参见第11章)或者改变系统接口的工作方式。这些选项的控制和可见性通常由一个(一些)内部寄存器决定。

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)也都集成了高速缓存本身。连最早期的 MIPS CPU 都在状态寄存器里有高速缓存控制的字段。 R4000 以后，就有专门的 CP0 指令来操纵高速缓存的每一项了。我们将在第 4 章讨论高速缓存。

例外/中断控制： 象中断或者例外时发生什么，您应该做什么来处理它等事情都由一些 CP0 控制寄存器和特殊指令来定义和控制。这会在第 5 章讨论。

存储管理单元控制： 第 6 章讨论这个话题。

杂项： 总是有更多的东西：时钟、事件计数器、奇偶校验错误检测等等。无论什么时候额外的功能被集成到 CPU 里边，不再能方便地当作外设访问时，这里就要增加一些东西。

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 MIPS 标准指令集缺少很多实际 CPU 需要的功能，但是它预留了多达 4 个的协处理器操作码和相应的指令域。其中一个(协处理器 1)时浮点协处理器，这的确是通常意义上的协处理器—原文： which really is a coprocessor in anyone's language 。
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，协处理器 0 指令是处理所有标准指令集范围之外的功能所必须的。这也是本章描述的对象。
协处理器 0 不能独立存在而且也绝不是可选的—例如，您不可能做一个没有状态寄存器的 MIPS CPU 。但它的确规定了访问状态寄存器的指令的编码方式。所以，虽然 R3000 和 R4000 家族的状态寄存器的定义发生了变化，您还是能用同样—译者：所谓同样，大概是指用同样的指令，具体的处理一般有所不同—的汇编程序来处理两种 CPU 。
协处理 0 的功能被有意地从 MIPS 指令集圈离开来，原则上是实现相关的。实际情况是这些功能和常规的指令集是配对发展的。例如，到目前为止制造的 MIPS III CPU 的 CP0 功能都非常相象，以致同样的操作系统二进制代码可以在整个家族的处理器上跑(可能需要稍微处理一下)。
四个协处理器中， MIPS III ,尤其是 MIPS IV 以后的"标准"指令集已经侵占了 CP3 。只有 CP2 还可以给一些片上系统应用使用。

我们会在本章后半部分总结所有在"标准" CPU 能找到的东西。但是让我们暂时别管我们想作到什么功能，先看看我们用什么机制吧。 MIPS CPU 里只有为数不多的几个 CP0 指令—只要可能，对 CPU 的底层控制都是对一些特殊 CP0 寄存器某些位的读写。

表 3.1 介绍了那些已经成为事实标准的控制寄存器功能描述。表中第一组的寄存器(及其功能)是到今天为止每个 MIPS CPU 都实现了的；第二组是自 R4000 (它代表着一次改善 CP0 部件组织方式的尝试)以后的 MIPS CPU 都实现了的。

这不是一个完整的列表；在讲到存储管理和高速缓存控制的时候我们将会看到更多一些控制寄存器。另外，一些 MIPS CPU 已经有一些和具体实现相关的寄存器—这也是往 MIPS CPU 里增加特色功能的标准方法。请参考您的特定 CPU 的手册。

为了防止这时候就用一堆的细节把您搞晕，我们把对 CP0 寄存器一位一位的描述放到不同的小节里：3.3 小节放所有 CPU 都有的寄存器；3.4 放 R4000 以后的 CPU 都有的寄存器。如果您对下面的章节感兴趣，现在可以暂时跳过那些小节。

我们列这些寄存器的时候， K0 和 K1 值得一提。那是两个由软件约定预留下来的通用寄存器，用在例外处理程序中。预留至少一个通用寄存器是非常必要的[1]；预留哪一个是硬性指定的，但必须保证所有的 MIPS 工具包和二进制程序都遵循同一约定。[2]

[1]译者：否则保存上下文时会有困难，因为 RISC 结构中所有的 load/store 都要通过通用寄存器执行，而且例外处理程序不能假定某个通用寄存器的值有效

[2]译者：这一段话多少有点跑题的感觉，不过考虑到 K0,K1 也是为系统控制服务的，也说的过去。要记住所谓 CP0 寄存器和一般可以参与运算的通用寄存器不同就是了。

表 3.1: 常见的 MIPS CPU 控制寄存器(不包括 MMU)

寄存器助记符	CP0寄存器标号	描述
PRId	15	识别这个处理器类型的一个标志符，带着更新版本号信息。这个 ID 原则上是应由 MIPS 公司控制的，指令集或者 CP0 寄存器集发生了改变的时候必须变化。到 97 年年中为止用过的值列表可以参见下面的表 3.2 。
SR	12	状态寄存器，罕见地由大部分可写的控制位域组成。包括决定 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	什么导致异常或者中断？
EPC	14	例外程序计数器：处理完例外/中断后从哪里重新开始执行。
BadVaddr	8	导致最近的地址相关例外的程序地址。各种地址错例外都会设置它，即使没有 MMU 。
Index	0	
Random	1	
EntryLo0	2	
EntryLo1	3	所有这些都是 MMU 操纵相关的寄存器，在第6章描述。 **EntryLo1** 和 **Wired** 是 R4000 引入的。
Context	4	
EntryHi	10	
PageMask	1	
Wired	1	
R4000 引入的寄存器		
Count	9	这两个寄存器一起形成了一个简单但是很有用的高精度时钟，频率为 CPU 流水线频率的一半。
Compare	11	
Config	16	CPU 参数设置，通常是系统决定；一些域可写，一些只读。
LLAddr	17	最近一次 ll(load-linked) 指令的地址。只用于诊断错误。
WatchLo	18	用于设置硬件数据观测点。可以在 CPU 存取这个地址时发生例外—可能对调试有用。
WatchHi	19	
CacheERR	27	当 CPU 在其数据通路上支持校验时，用于分析(甚至可能从中恢复)一个内存错误。详细信息参见图 4.4 和它的解释。
ECC	26	
ErrorEPC	30	
TagLo	28	用于高速缓存操纵的寄存器，详见 4.10 小节。
TagHi	29	

3.1 CPU 控制指令

有几条 CPU 控制指令用于实现存储管理，但我们把它留给第 6 章。 MIPS III CPU 有个多功能的 **cache** 指令来做所有对高速缓存的操作，第 4 章会进一步说明。但除此之外， MIPS CPU 控制还需要少数几个指令。首先看看用来访问刚刚我们列出的那些寄存器的指令：

```
mtc0     rs, <nn>  # 把数据送到协处理器0
dmtc0    rs, <nn>  # 把双字数据送到协处理器0
```

这些指令把通用寄存器 **rs** 的内容装到协处理器0寄存器 **nn** ，数据分别位 32 位和 64 位(即使在 64 位的 CPU 里，很多 CP0 寄存器也是 32 位的)。这是设置 CPU 控制寄存器的唯一方法。

直接在汇编程序里使用控制寄存器的编号来引用它们是不良习惯；通常您应该使用如表 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；您的工具包文档会告诉您如何做。虽然原始的 MIPS 标准有很强的影响，但是(不同的工具链中)这些寄存器的命名还是有所差别。我们将一直使用表 3.1 中的助记符。

与之相反的是从 CP0 控制寄存器中取出数据：

```
mfc0     rd, <nn>  # 从协处理器0取出数据
dmfc0    rd, <nn>  # 从协处理器0取出双字数据
```

在两种情况下通用寄存器 **rd** 都被装入 CPU 控制寄存器 **nn** 的值。这是查看一个控制寄存器值的唯一方法。因此，如果您想要更新控制寄存器的某个域，比如说状态寄存器 **SR** 吧，您写的代码将是这个样子：

```
mfc0     t0, SR
and      t0, <要清掉的位的补码>
or       t0, <要设置的位>
mtc      SR, t0
```

控制指令集的最后一个关键成员是一种取消例外效果的方法。我们会在第5章详细讨论例外的问题，但基本的问题是每个实现任何一种安全操作系统的 CPU 都要面对的；那就是例外可以在运行在用户态(低特权级)时发生，而例外处理程序运行在高特权级。因此当返回用户态时，CPU 需要避开两种风险：一方面，如果在返回用户程序之前特权级降低了，您马上就会得到一个致命的特权级违反例外[3]；另一方面，如果先回到用户态再降低特权级，那么一个恶意的程序就有可能有机会用高特权级运行指令。所以返回到用户程序和降低特权级必须是从编程的角度不可分的操作(或者用体系结构术语说，原子的(操作))。

在 R3000 和类似的 CPU 中，这个工作是由一个延迟槽放一条 **rfe** 指令的跳转指令来完成的；但从 R4000 以后，**eret** 完成整个事情。第 5 章里我们会更详细的谈到它们。

[3]译者：因为至少还有一些属于例外处理程序的特权级指令需要运行

3.2 起作用的寄存器及其时机

有些寄存器您需要在下面这些情况和它们打交道：

- 加电后： 您需要设置 **SR** 来使 CPU 进入正确的引导状态。

  绝大部分的 MIPS CPU (除了最古老的一些)都有 **Config** 寄存器，它可能包含一些需要在很早的时候设置的选项。请和您的硬件工程师商量，确认 CPU 和系统关于配置的问题足够一致，至少能启动到让您写这些寄存器!

- 处理任何例外： 任何 MIPS 例外(除了一个特别的MMU事件[4])都调用一个固定入口地址的“通用异常处理程序”。

  在入口处程序的寄存器并没有被自动保存，只有返回地址被存在 **EPC** 寄存器。MIPS 硬件没有任何关于栈的知识。在任何情况下一个安全操作系统的特权级例外处理程序不能假定用户级代码的任何完整性—特别地，它不能假定栈指针有效或者栈空间可用。

  您需要用 **K0** 和 **K1** 中至少一个来指向为例外处理程序预留的一些内存空间。然后您就可以保存东西，必要时还可以用另一个来访问控制寄存器。

  通过 **Cause** 寄存器，您可以找出例外的类型，再分别处理。

- 从异常处理返回： 控制最终必须返回到刚近入例外时保存的EPC指向的地方。不管发生的是什么例外，您返回时都要把 **SR** 寄存器设置会原来的值，恢复用户特权级设置，使能中断，也就使要消除例外的影响。

  在 R3000 中特殊指令 **rfe** 做这件事情，但是请注意它本身并不转移控制流。要跳回去，您要把原来的 **EPC** 值装到一个通用寄存器，然后用一个 **jr** 操作。

  在 R4000 和目前为止所有的64位CPU中，“从例外返回”指令 **eret** 接合了返回到用户空间和重新设置 **SR** 寄存器两个功能。

  严格地说， CP0 指令集，包括 **rfe** 和 **eret** ，都是实现相关的。但没有一个 CPU 用了第三种方法来做这个事情，假定以后也没有人会是相当安全的。然而，以后您可能会看到一个32位的 CPU ，它的 CP0 设计是基于 R4000 的[5]。

- 中断： **SR** 用来调整中断掩码，即决定哪些(如果有的话)中断被赋于比当前优先级更高的优先级。硬件没有提供中断优先逻辑，但是软件可以随便干。

- 总是触发例外的指令： 这些指令很常用(系统调用，断点以及模拟一些指令等)。所有的 MIPS CPU 都实现了 **break** 和 **syscall** ；有一些还实现了额外的一些指令。

[4]译者：指 TLB refill 例外，实际上后来的 CPU 还有几个特殊入口，不过用得不多，可以不管
[5]译者：龙芯-1就是，呵呵

控制寄存器编码: 关于保留域的一个说明现在有必要了。许多不用的控制寄存器域被标记为“0”；在这样的域里的位保证读出为0，写它也没有什么害处(虽然写入的值会被丢弃)。另一些被标记为“x”；您应该小心，保证总是写入 0，而且不应该假设读回的值是 0 或者其它任何特殊值。

3.3 标准 CPU 控制寄存器编码

这一节告诉您控制寄存器的格式以及各个域的一个概要功能描述。多数情况下，关于这些东西如何工作的更多内容在后面几节可以找到。但我们把有关存储管理的寄存器留到第 6 章。

3.3.1 处理器 ID*(PRId)* 寄存器

图 3.1 显示了 **PRId** 寄存器的内容。它是一个标志 CPU 类型的只读寄存器。只要指令集或者控制寄存器定义发生改变，“Imp” 就会改变。“Rev” 完全取决于制造者，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。我们所知道的一些设置列在表 3.2 中。

如果您想打出这些值，打成“x.y”的形式比较方便(其中 x,y 分别为 Imp 和 Rev 的十进制值)。尽量不要依赖这个值来获得一些参数(例如高速缓存大小，速度等等)或者获得某项特性是否存在的信息；用一些代码序列来探测各种特性的存在性，它将使您的软件更加可移植和健壮。很多情况下您会在本书找到(关于探测的)例子或者建议。

3.3.2 状态寄存器 *(SR)*

MIPS CPU 有少数几个模式位，它们在状态寄存器中定义，如图 3.2 。我们显示了“标准”的 R3000 和 R4000 CPU 的寄存器定义；其它 CPU 偶然也用其它域，或者改变一些域的含义，通常它们并不实现所有的域。

我们再次强调， MIPS CPU 里没有“nontranslated”(不经过TLB地址翻译)或者“noncached”(不缓存)模式；所有的是否翻译，是否缓存都由程序的地址决定。

绝大部分 MIPS CPU 都提供 R3000 和 R4000 所公有的那些域。

***R3000* 和 *R4000* 公有的关键域**

这是关键的公有域；把这些域重用为其它任何目的都是非常不好的想法，在可以预见的将来这些域的用法很可能都不会变化。

CU1 协处理器 1 可用：如果有浮点处理部件的话，设成 1 表示可以使用它；0 表示禁止使用。当值为 0 时，所有浮点指令导致例外。没有浮点硬件时把它设为 1 显然不行；但有浮点硬件时(用 0)关掉它有时会有用。[6]

[6]为什么要关掉一个好好的浮点部件呢？有些操作系统对所有的新任务禁止浮点指令；如果该任务试图使用浮点时，操作系统会捕获到例外并为它使能浮点部件。这样，我们可以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，这样可以节省上下文切换的时间。

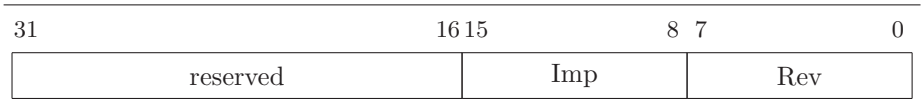


图 3.1: **PRId** 寄存器各个域

表 3.2: MIPS CPU 的 **RPID(Imp)** 值

CPU 类型	***Imp*** 值
R2000	1
R3000,IDT R3051,R3052,R3071,R3081. 绝大多数是早期32位 MIPS CPU	2
R6000	3
R4000,R440	4
一些 LSI Logic 的32位 CPU	5
R6000A	6
IDT R3041	7
R10000	9
NEC Vr4200	10
NEC Vr4300	11
R8000	16
R4600	32
R4700	33
R3900和其变种	34
R5000	35
QED RM5230,RM5260	40

位 31 和 30 分别控制协处理器 3 和 2 的可用性；可能被一些想定义更多指令的 CPU 使用。 CP2 指令可能出现在一些(用于 SOC 的?)处理器核的实现中。

BEV 启动时例外向量：当 BEV==1 时， CPU 用 ROM(KSEG1) 空间的例外入口(参见5.3节)。正常运行中的操作系统里，BEV 一般设置为 0 。

IM 中断屏蔽：8 位，定义那些中断源有请求时可以触发一个例外。八个中断源中 6 个是 CPU 核外面的信号产生的(其中一个可以被浮点部件使用；它虽然在片上，逻辑上是外部的)；其它两个是 **Cause** 寄存器中软件可写的中断位。

有浮点部件的32位 CPU 用 CPU 中断之一来发出浮点例外[7]；MIPS III和以后的处理器协处理器 0 中通常有个内部时钟，时钟中断信号通过最高的中断位来发出。其它情况，中断从 CPU 片外发出。

这里没有为您提供中断优先逻辑：硬件对所有中断位一视同仁。详细信息参见 5.8 节。

[7]译者：也不尽然，龙芯-1就不是这样

R3000(MIPS I) 状态寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	8	7	6	5	4	3	2	1	0
0		CU1	CU0	0		RE	0		BEV	TS	PE	CM	PZ	SwC	IsC	IM		0		KUo	IEo	KUp	IEp	KUc	IEc

R4000(MIPS III) 状态寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	8	7	6	5	4	3	2	1	0
0		CU1	CU0	RP	FR	RE	0		BEV	TS	SR	0	CH	CE	DE	IM		KX	SX	UX	KSU		ERL	EXL	IE

图 3.2: **status**寄存器各个域

不那么明显的公有域

这些域比较生僻，通常不用，但不好随便改变，因此目前为止都一致。

CU0 协处理器 0 可用：设成 1 允许用户态下使用一些特权指令。您不会想这么干的。协处理器 0 的指令在内核态总是可用的，不管这个位设为什么值。

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 11.6 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倾向于大尾端，但现在已经彻底混淆了。

这个“世界”的操作系统能运行另一个“世界”来的软件可能会是一个有用的特性； RE 位使得这成为可能。当 RE 设为 1 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，真的达到跨世界运行会需要软件上很大的努力，到目前为止还没有干过。

TS TLB 关闭：详细的信息参见第 6 章。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，在这种状态下继续操作有可能导致内部竞争损坏芯片，所以 TLB 停止匹配任何地址。 TLB 关闭是一个终结性的过程，一旦置上只有硬件复位才能清除。

一些 MIPS CPU 的 TLB 硬件可以防止出现这种情况，因而可能并不实现这一位。

在 IDT R3051 系列 CPU 中，您可以在硬件复位后查看这一位，它当且仅当 CPU 没有TLB(存储管理硬件)的时候置位。但这种测试并不是总是可靠的(即有些硬件实现可能并不是这样做)。

状态寄存器中 ***R3000*** 专有的域: 日常使用的

Swc,IsC 交换高速缓存和隔离(数据)高速缓存：这些是为了高速缓存管理和诊断用的高速缓存模式位；详细信息参见 4.9 节。简单地说，当 **SR(IsC)** 置位时，所有的 load 和 store 只访问高速缓存，绝不访问内存；在这种模式下一个部分字的 store 操作将使相应的高速缓存表项无效。

当 **SR(SwC)** 置位时，指令高速缓存和数据高速缓存的角色互换，这样您可以访问和无效指令高速缓存的内容。

KUc,IEc 这是两个基本的 CPU 保护位。

以内核优先权运行时， KUc 设成 1 ，用户模式下设成 0。在内核模式下您可以访问整个程序地址空间以及使用特权(协处理器 0)指令。在用户模式下您只能存取 0—0x7FFFFFFF 之间的程序地址，不能使用特权指令；试图违反规则会导致例外。

IEc 设置为 0 阻止 CPU 响应中断，1 使能中断。

KUp,IEp 上一个KU,上一个IE：例外时，硬件把 KUc 和 IEc 的值保存在这儿，再把它们设置为 [1，0] (内核模式，禁止中断)。 rfe 指令可以用于把 KUp,IEp 拷贝回 KUc，IEc 。

KUo,IEo 老的KU,老的IE：例外时，硬件把 KUp,IEp 的值保存在这儿。效果上这六个 KU/IE 位构成了一个三项每项两位的栈，例外时压栈，rfe 时弹栈。这个过程在第5章描述并展示在图 5.1 中。

如果在一个例外处理程序保存 **SR** 寄存器之前又发生了例外，这种机制就使得我们有可能干净地处理嵌套的那个例外。这种情况下能做的事情是很有限的，很可能它只是对把 TLB 重填的代码写短些有用；更多的信息请参见6.7节。

生僻的 *R3000* 专有位

PE 当一个高速缓存奇偶校验错误发生时置位。这种情况下不发生例外，只是对诊断问题有用。之所以 MIPS 体系结构有高速缓存诊断的设施是因为早期的 CPU 使用片外的高速缓存，而高速缓存总线上的信号时序已经接近当时工艺水平的极限。对那些实现来说，高速缓存的奇偶校验位是很必要的调试工具。

对拥有片上高速缓存的 CPU 来说，这个特性很可能已经过时。

CM 这显示了数据高速缓存“隔离”以后最后一个 load 操作的结果(关于“隔离”的意思，请参见 IsC 位的解释或者 4.9.1 节)。如果高速缓存真的包含被访问地址的数据(也就是说，即使数据高速缓存没有被“隔离”，访问也将命中)，那么 CM 将被置位。

PZ 当它置位时，高速缓存奇偶校验位被写为 0 ，不再进行奇偶校验。这是使用片外高速缓存的 CPU 用的老古董了。它可以让有信心的设计者省去保存奇偶校验位的外部存储器，节约一点钱。如果 CPU 有片上高速缓存，您用不着这一位。

R4x00 CPU 中常见的域

请记住，这些域原则上是完全 CPU 相关的；然而， MIPS III 以上的 CPU 都有很多相同的地方。

FR 一个模式开关：设成 1 使得所有32个双字大小的浮点寄存器对软件可见；设成 0 使它们象在 R3000 上那样工作[8]。

[8]译者： 32 位 MIPS 处理器用一对 32 位寄存器来存一个双精度浮点数，参见第 7 章

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 的设计者要费这功夫去设计一个从来没有人用过的特性呢?

在 1989-90 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 大概比 Alpha 早18个月面世。不管 DEC 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；而显然 VMS 的体系结构设计师声称只有两个特权级不可能实现 VMS 。

Alpha 的基本指令集和 MIPS 几乎完全相同；它的最大不同是试图取消子字存取操作，后来的 Alpha 指令集又重新加回了那些指令。

最后，看起来 VMS 软件组选择了 Alpha 而不是 R4000，因为它坚持认为某些指令集和 CPU 控制结构的不同会使得移植到 R4000 慢很多。我很怀疑这个说法 (and put the choice down to NIH(not invented here)—不会翻 。DEC 相信控制它自己的处理器开发很重要，这很可能是对的，但猜猜如果 DEC 采用了 R4000 事情会怎样发展也很有趣。

我也怀疑卖出的基于 Alpha 的 VMS 几乎可以忽略，但那是另一回事了。

SR 发生了软复位： MIPS CPU 提供了几个不同等级的复位，用硬件信号区分。**SR(SR)** 域在硬复位(这时所有的参数都重新设置)后被清掉，在一个软复位或者不可屏蔽例外后置位。特别地，配置寄存器**Config** 在软复位期间维持原值，但硬复位后必须重新编程。

DE 禁止高速缓存和系统接口的数据检查：一些硬件系统可能没有在高速缓存重填的路径上提供奇偶校验(虽然硬件设计者可以选择把返回给 CPU 的数据标记为没有校验位—这很可能是更好的方法[9]，这时您可能要设置这一位。对没有实现高速缓存奇偶校验的 CPU，您也应该设置这一位。

UX,SX,KX 这些用于支持 R3000 兼容的和一些扩展的地址空间：三个不同的特权级各有一位；当相应的位置位时，最常见的内存地址翻译例外(即 TLB 不命中例外)被重定向到不同的入口，那里的软件将处理64位的地址。

同时，当 **SR(UX)** 置成 0 时 CPU 将不在用户态下运行 MIPS III 中的 64 位指令。

KSU CPU 特权等级： 0 是核心态，1 是管理态，2 是用户态。不管这个域是什么值，只要 EXL 或者 ERL 被例外置位了， CPU 就自动处在核心态。管理态是 R4x00 引入的，但从来没有被用过。(猜测的)原因可以参见边栏。

ERL 错误级：当 CPU 响应一个奇偶校验或者 ECC 校验错误例外时被置位。之所以这个要用一个单独的位是因为一个可以纠正的 ECC 错误可以在任何地方发生—包括最敏感的一般例外处理代码—如果系统想修正 ECC

[9]译者：大概是指这样高速缓存部件可以自动禁止检查奇偶检验

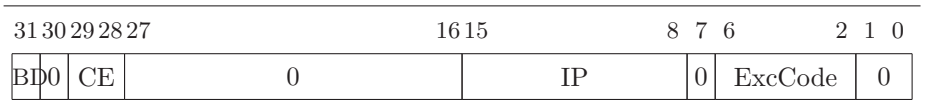


图 3.3: **Cause** 寄存器各个域

错误并继续运行，它必须不管例外发生在哪里都可以修复。这是有挑战性的，因为例外处理程序没有一个可以安全使用的寄存器；而没有一个寄存器用做指针，它就无法开始保存寄存器。为了跳出这个死圈，**SR(ERL)** 有一个很彻底的效果；所有对正常用户地址空间对访问消失了，从 0 到 0x7FFF.FFFF 的地址变成一个映射到相同物理地址的不经过高速缓存的窗口。目的是高速缓存错误例外处理过程可以用 0 号寄存器(值永远为 0)来做基地址，用基址+偏移的方式来获得一块可以用来保存寄存器的内存空间。

EXL 例外级：被任何例外置位，这强制进入核心态并禁止中断；目的是把 EXL 维持足够长的时间以便软件决定新的 CPU 特权级和中断屏蔽位该设成什么。

IE 全局的中断使能位：请注意不管这怎么设， EXL 或 ERL 总是禁止所有的中断。

***R4x00 CPU* 里的 *CPU* 相关域**

RP 减小功耗：降低 CPU 的操作频率，通常是把它除以 16 .在很多 R4x00 CPU 里这不起作用；即使起作用，它也要求系统接口也能对付这种要求。具体情况请阅读 CPU 手册，咨询系统设计人员。

CH 高速缓存命中指示：只用于诊断。

CE 高速缓存错误：这只对诊断和错误恢复过程有用，错误恢复也应该依赖 **ECC** 寄存器里的内容而不是这。

3.3.3 原因寄存器 *(Cause)*

图 3.3 显示了 **Cause** 寄存器各个域，这是您想找出发生了什么例外，决定如何处理时应该看的东西。**Cause** 寄存器是例外处理的一个关键寄存器，在我所知道的 MIPS CPU 中定义都一样，只是其中例外类型的列表有所增长。

BD 转移延迟：**EPC** 寄存器作用是保存例外处理完之后应该回到的地址。正常情况下，这指向发生例外的那条指令。但是如果发生例外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，但如果您返回到延迟槽指令，转移指令将没法跳转从而这个例外将破坏程序的执行。

Cause(BD) 只当发生例外的指令在转移指令延迟槽时置位。如果您想分析发生例外的指令，只要看看 **Cause(BD)** (如果它为 1 ，那么该指令是 **EPC**+4)。

CE 协处理器错误：如果例外是由于一个协处理器格式的指令没有被相应的 **SR(CUx)** 位使能引起的，那么 **Cause(CE)** 保存这条指令的协处理器号。

IP 待决的中断：展示想要发生的中断。第 7到2 位随着 CPU 六个中断输入的电平变化。第8位和第9位可读可写，保存您最后写入的值。当这 8 位任何一个活跃而且被 **SR(IM)** 位和全局中断标志 **SR(IEc)** 使能时，一个中断将被触发。

Cause(IP) 和 **Cause**寄存器其它域有微妙的不同：它不是告诉您当例外发生时发生了什么事情，而是告诉您现在正在发生什么事情。

ExcCode 这是一个5位的代码，告诉您哪种例外发生了，如表3.3所示。

3.3.4 例外返回地址 *(EPC)*

这只是一个保存例外返回点的寄存器。一般等于导致(或者遭受)例外的指令地址，除非 **Cause** 寄存器的 BD 位置位了—这种情况下 EPC 指向前一条(转移)指令。如果 CPU 是 64 位的那么 EPC 也是。

3.3.5 无效虚地址寄存器 *(BadVaddr)*

这个寄存器保存引发例外的地址；在任何 MMU 相关的例外里设置，原因包括一个用户程序试图访问 kuseg 以外的地址，或者地址没有正确对齐。在其它任何例外之后它的值没有定义。请注意，特别地，总线错例外并不设置它。如果 CPU 是 64 位的那么 EPC 也是。

表 3.3: ExcCode 值：不同种类的例外

值	助记符	描述
0	Int	中断
1	Mod	TLB 修改：试图写一个经过 TLB 映射的程序地址，但 TLB 表项说那是只读的—译者：原书似乎有误。
2 3	TLBL TLBS	TLB load/TLB store：读/写使用的程序地址在 TLB 里没有匹配的项。这个例外有一个专门的入口，用来处理大部分的地址翻译(它们就是从 R3000 到 R4000 的改变中获得特殊对待的例外)。
4 5	AdEL AdES	地址错(分别是取指/ load 操作和 store 操作引起)：要么是在用户态下试图访问 kuseg 以外的段，或者是试图访问一个双字、字或者半字而地址不相应对齐。
6 7	IBE DEB	总线错误(分别是在取指或读数据时发生)：外部硬件指示发生了某种错误；您该怎么做是系统相关的。存数操作引起的总线错只能间接地反应出来，表现为读入想写的高速缓存块时的结果。
8	Syscall	由一个 **syscall** 指令无条件产生。
9	Bp	断点：由 **break** 指令产生。
10	RI	保留指令：一条本 CPU 没有定义的指令。
11	CpU	协处理器不可用：一种特殊的未定义指令例外。指令属于某个协处理器或者协处理读写指令。特别地，这是当浮点部件可用位 **SR(CU1)** 没有置位时浮点指令引起的例外，因此它也就时浮点模拟开始的地方。
12	Ov	算术溢出：请注意无符号类的指令(如 **addu**)从不引起这个例外。
13	Trap	这个来自 MIPS II 新增的条件陷阱指令。
14	VCEI	指令高速缓存中的虚地址一致性错误：这个只和有二级高速缓存并且使用二级高速缓存的 tag 位来检查高速缓存别名的 R4000 以后的 CPU 相关。 4.14.2 节有相关解释。
15	FPE	浮点例外：只在 MIPS II 和它以上的 CPU 中发生。在 MIPS I CPU 中，浮点例外作为中断发出。
16	C2E	协处理器 2 例外：还没有一个 R4x00 CPU 有协处理器 2 ，所以不必管它。
17-22	-	预留作将来的扩展。
23	Watch	load/store 的物理地址和 **WatchLo/WatchHi** 寄存器中的值匹配。
24-30	-	预留作将来的扩展。
31	VCED	数据虚地址一致性错误：和 VCEI 一样。

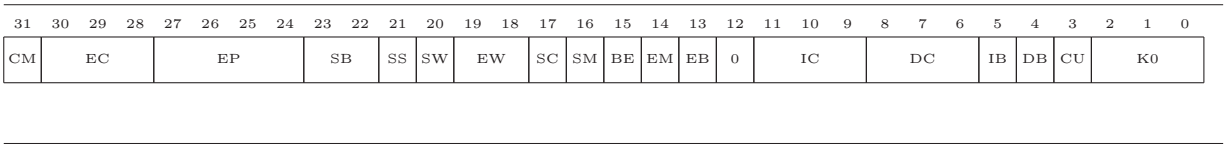


图 3.4: **Config** 寄存器各个域

3.4 R4000 以后的 CPU 专有的控制寄存器

R4000 (第一个实现了 64 位 MIPS III 指令集的 CPU)是一个相当大胆的尝试。它试图把当时已经有些控制不住的各种实现方式规则化，并给一些不可避免的特色功能(的实现)提供一个规则的结构。

最明显的改变是高速缓存现在是由一条叫 **cache** 的新指令控制(实际上是一组指令)；其它而外的特色功能包括 CPU 内带的时钟，一些调试设施和处理高速缓存的可恢复位错误的机制。同时提供一个 **Config** 寄存器来允许一些关键特性的参数化(高速缓存总容量， cache 行大小等)，软件可以通过它来进行相应控制。

我们将在第 4 章介绍那些只用于高速缓存管理的寄存器，在第 6 章介绍 MMU/TLB 寄存器。

3.4.1 *Count/Compare* 寄存器： *R4000* 时钟

这些寄存器提供了一个简单的连续运行的通用时钟，可以编程来发出中断。在大部分的 CPU 里，这个时钟是不是连线到一个中断是复位时的一个选项。时钟中断总是使用 **Cause(IP7)** (通常这使得硬件输入 Int5*多余了–译者：应该是不能用了？)。

Count 是一个32位的计数器，它精确地以 CPU 流水线频率的一半向上加(即每两拍加 1) 。当它达到最大的32位整数值时，直接溢出回 0 。您可以读 **Count** 寄存器来获取当前时间。您也可以随时写 **Count** 寄存器，但实践上还是不这么做为好。

Compare 32位可读可写的寄存器。当 **Count** 寄存器增长到等于 **Compare** 寄存器时，中断就回发出。这个中断一直维持到下一个对 **Compare** 寄存器的写为止。

要产生一个周期的中断，中断处理程序应该总是用一个固定数量来递增 **Compare** 寄存器(不是 **Count** ，因为那样的话中断处理的延迟会稍微增加周期时间)。软件需要看看一个中断是不是来迟了，以避免把 **Compare** 寄存器设置成一个 **Count**已经经过的值。通常，它写完 **Compare** 后再重新读 **Count** 以检查这个问题。

3.4.2 *Config* 寄存器： R4x00 配置

CPU 配置毫无疑问地是 CPU 相关的，但所有 R4x00 家族地成员都有 **Config** 寄存器并共享其中的许多域。图3.4显示了最初的 R4000 CPU 提供的标志位集合。

图3.4中的域如下：

CM 设为 1 表示主设备/检查器 (?) 模式—只用于容错系统。在复位时设置，只读。

EC 三位，用于表示时钟分频：内部流水线时钟和系统接口时钟的比率。在一些 CPU 里，系统接口时钟等于输入时钟，然后这作为乘数(倍频后)提供给内部时钟；在老一些的 CPU 里，流水线频率总是等于输入时钟的两倍，然后这作为被除数(分频后)算出系统接口时钟。

对于 R4000 ，当这个域的值等于 n 时，比率是 (n+2) 。但后来的 CPU 中诸如 1.5 和 2.5 这样的比率使得编码不得不改变。请参照具体的 CPU 手册。

这个域(到目前为止)在复位时设置，只读。

EP 四位，用于表示数据传输模式。 R4000 和以后的许多处理器的系统接口都没有为高速缓存写回时的多块数据传输提供外部握手信号。 CPU 能够每拍发送一个宽度等于总线宽度的数据。因为这有时对接口来说太快了，所以数据传输的速率和节拍在这里编程控制。

下面的表显示 “D” 时表示一个发送了一个数据字的拍，显示 “x” 时表示系统接口歇一个时钟周期。

EP值	数据模式	EP值	数据模式
0	D	8	Dxxx
1	DDx	9	DDxxxxxx
2	DDxx	10	Dxxxx
3	Dx	11	DDxxxxxxx
4	DDxxx	12	Dxxxxx
5	DDxxxx	13	DDxxxxxxx
6	Dxx	14	Dxxxxxx
7	DDxxxxx	15	DDxxxxxxxx

短的模式必要时重复，因此一个8个字(4个双字)的高速缓存块，当 **Config(EP)** 等于5时将是 “DDxxxxDD”。(还是正确的写法是 “DDxxxxDDxxx” ，表示总线上有三个空闲周期？)我们的经验是许多 CPU 在写结束是不实现无用的周期(dead time)，但有一些确实这样做了。如果这对您很重要，去问您的 CPU 提供商吧。

大部分 CPU 只支持这些值的一个子集。有些使用不同的编码。这个域有时在复位时设置并只读，有时是可编程的。

SB 片外二级高速缓存块大小(或者说行大小)。这个域通常由硬件决定，是只读的。 R4000 的编码是：

SB值	块大小 (32位字)
0	4
1	8
2	16
3	32

SS 在 R4000 CPU 中，片外的二级高速缓存可以是分立的(指令和数据分别使用不同的高速缓存位置，不管地址是什么)或者统一的(指令和数据根据地址统一对待)。1 表示分立，0 表示统一。

SW 在 R4000 (也许还有其它) CPU 中，如果二级高速缓存是和原始的 R4000SC 一样位 128 位宽则设置为 1 ，0 表示 64 位宽。

EW 系统接口宽度：0 表示 64 位，1 表示 32 位。

SC 在 R4000 和 R5000 以及它们的直接后代中，这个域是可写的，作为软件控制的二级高速缓存使能位；它对诊断问题非常有用。如果有一个片上控制的二级高速缓存，这个设置为 1 ，否则为 0 。

后来的一些带二级高速缓存的单处理器在另外的域(利用 R4000 用于多处理器的一些域)中报告二级高速缓存的大小。然而，通常这些大小域的值只是机械地传递加电配置时收到的信息，并没有硬件上的影响。[10]

SM 多处理器高速缓存一致性协议配置。

BE CPU 尾端(参见 11.6 节)： 1 表示大尾端，0 表示小尾端。在(至少) NEC Vr4300 这个域时软件可写的，但在大部分 CPU 上它是硬件配置的一部分。

EM 数据校验模式： 1 表示 ECC 校验，0 表示每字节的奇偶校验。

EB 一定是 0 。曾经想提供一个硬件接口选项来用顺序次序进行高速缓存重填和写回操作，而不是子块次序；这个选项从来没有实现过。

IC/DC 一级指令/数据高速缓存的大小：一个二进制值 n 表示高速缓存大小为 2^{12+n} 字节。

IB/DB 一级指令/数据高速缓存的行(块)大小：0 表示 4x32 位字， 1 表示 8x32 位字。

CU 另一个多处理器高速缓存一致性协议配置位。

K0 这是一个可写的域，用来配置 KSEG0 段访问的高速缓存行为。使用的编码和 MMU 表里控制每一页的缓存行为用的 **EntryLo(c)** 位一样。除了多处理器一致性用的值之外，我们感兴趣的只有 3 =缓存， 2 =不缓存。

R4000 以后的不提供多处理器高速缓存支持的 CPU 已经用一些其它值来配置不同的高速缓存行为，例如写穿透和写分配—它们的含义请参见 4.3 节。

3.4.3 *Load-linked Address (LLAddr)*寄存器

这个寄存器保存最近运行的 load-linked 操作的物理地址，用来监控可能导致后来的一个条件存 (store conditional) 失败的访问；参见 5.8.4 节。软件对 **LLAddr** 的访问只用于诊断目的。

[10]译者：我不明白这一段和 SC 有什么关系。

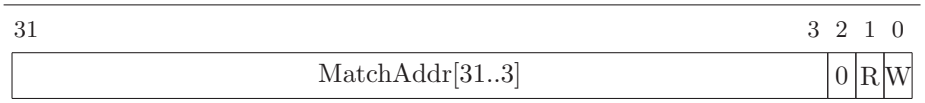


图 3.5: **WatchLo** 寄存器各个域

3.4.4 调试观测点 *(WathLo/WatchHi)* 寄存器

这对寄存器实现了一个观测点：它们包含一个物理地址，每个读数据或者存数据操作都跟它比较，如果地址匹配则发生一个陷阱例外。目的是给调试软件提供帮助。

WatchLo 显示在图3.5中。观测点的地址只维护到最近的双字(8 字节)，所以只有第三位以上的地址位需要保存。 **WatchHi** 保存地址高位。其它的 **WatchLo** 位如下：如果 **WatchLo(R)** 等于 1 ，读操作参与检查，如果 **WatchLo(W)** 等于 1 ，存数操作参与检查。您完全可以同时使能读操作和存数操作的检查。

有些调试器使用硬件观测点，有些则不用。提供观测点(有时叫数据断点)功能的调试器通常允许您设置任意多个这类断点，很可能只有您指定的调试点正好是一个时才会使用 **WatchLo/WatchHi** 寄存器。

See MIPS Run

翻译：Alan Yao

10 MIPS 上的 C 语言编程

本章主要讨论用 C 语言建立完整的 MIPS 系统可能需要具备的一些知识，因此，更多时候本章讲述 C 编译器产生的汇编语言代码，而不是 C 语言代码。为避免讨论过于繁琐，而使本章膨胀到一本新书的规模，现假定读者您是第一次向 MIPS 平台移植代码。

一个高效的 C 运行环境依赖于 C 语言程序的寄存器使用约定，这一般由 C 编译器强制规定，因此对于汇编工程师来说，也是需要强制遵守的。参照 2.2.1 部分对寄存器使用的全部约定，本章内容涉及：

- 堆栈、子程序链接、参数传递：关于 MIPS 进程是如何实现的，以及如何为避免不必要工作而支持的各种特性
- 共享库和非共享库：关于在复杂机器上支持共享库 OS 的一点注解
- 介绍编译器的优化：可能对 MIPS 上 C 语言编程造成的影响
- C 语言访问设备的提示：关于如何写绝大多数设备驱动

即使你使用其他的高级语言而非 C 语言，只要你想为 MIPS 编译代码、并与标准库链接，那么本章的大多数内容，还是对你有所帮助的。在这儿，我并没有针对特定编程语言，是因为我对他们了解不够，一直不知道如何恰当的点到为止。

10.1 堆栈、子程序链接、参数传递

许多 MIPS 程序使用混合语言编写的——对于嵌入式系统的程序员，这最可能是在 C(或 C++)中加入汇编语言。

一开始，MIPS 社团建立了一套约定，用来规范如何传递参数给函数（在 C 语言中，称为“子例程”或“子程序”）和从函数返回值。这个约定看起来很复杂，其实只是为了逻辑上遵循文档规则，而使文档太过庞大而已。

基本原则是所有参数在堆栈中的一个数据结构中分配空间，只有少数堆栈开始部分的内容可以装入 CPU 寄存器——相应的内存空间将变得是没有定义的。实际上，这意味着对于大多数调用，参数全部传递到寄存器中；然而，堆栈数据结构是理解进程的最好切入口。

自从 1995 年 Silicon Graphics 开始，已经为了提高性能而对调用约定作了修改。并对这些修改作如下命名：

- o32：传统的 MIPS 约定（o 是 old 简写）；详细说明如下。这个约定（不包括 SGI 为支持共享库而添加的一些特性）目前还是嵌入式工具相当常用的；不过过不了多久，两个最新的约定将会被其他工具作为可选项加以支持。
- n64：针对 64 程序的约定。SGI 的 64 模式意味着指针和 C 语言的 long 整数类型都是 64 位的。对于嵌入式应用程序，更长的指针意味着越界，这样会产生疑惑：这个约定现在是否用在了工作站环境上？不过，n64 改变了使用寄存器的约定和参数传递规则，因为 n64 将更多的参数放进了寄存器，从而提高了性能。
- n32：在参数传递上采用了 n64 的规则，不过指针和 C 语言的 long 整数类型都是 32 位的。然而，SGI 和其他编译器支持扩展的 long long 整数类型，从而实现硬件支持的 64 位整数。这个编译模式在嵌入式系统变得很流行

这里先描述 o32 标准，然后指出 n32 和 n64 的差异部分（10.8 节）。

这本书出版时，还有其他有争议的标准，不过大多数和 MIPS EABI 相类似。所有的 MIPS EABI 项目的目的是要产生一个范围更广的标准，以便嵌入式工具能相互工作的更好。这是个绝好的主意，不过这个新的调用约定是从类似 SGI n32 的私人项目继承来的（虽然简单，但没有很好的兼容性）。我们虽然很希望这能产生很好的结果，但是目前在嵌入式应用中合理使用 o32 编译模式，也不会失去什么。

10.2 堆栈参数结构

从这节开始，将陆续介绍 SGI 称为 o32 的原始 MIPS 约定。并在 10.8 节才开始明确介绍新约定变化。

MIPS 硬件不直接支持堆栈，但是调用约定需要。堆栈是向下延伸的，当前堆栈的底保存在寄存器 sp($29)中。任何提供保护和安全的 OS 都不支持用户堆栈，而且除非在函数调用的地方，sp 的值没有价值。但是约定还是在函数使用的堆栈的最下方保留了 sp。

在函数调用的地方，sp 必须是 8 字节对齐的（对于 32 位 MIPS 硬件，不是必须的；但是对于 64 位 CPU，是必须的）。子程序总是将堆栈指针调整为 8 的倍数，然后填到 sp 中。(注：SGI 的 n32 和 n64 标准调用的堆栈都是以 16 字节对齐的方式维护的)

在 MIPS 标准中，为了调用子程序，调用者在堆栈中建立一个数据结构来保存参数，并将 sp 指向这个数据结构。第一个参数（在 C 程序中位于最左的）在内存中是处在最下方。每个参数至少占据一个 word（32 位）；64 位的值，比如浮点 double 类型和(对于某些 CPU)64 位整数，必须是 8 字节对齐的（就好像是个包含 64 位“纯量场 scalar field”的数据结构）。

参数结构就像一个 C 的 struct，但是会有更多的规则。首先，为任何调用分配一个至少 16 字节的参数空间，即使没有参数。其次，char 和 short 等比 word 类型短的数据类型，以一个 int 类型（32 位）传递。不过，这种处理方式不能用于 struct 内部。

10.3 使用寄存器传递参数

任何位于参数结构开头 16 字节内的参数都被传递到寄存器中，调用者可以不明确定义参数结构中的这 16 字节。存在于堆栈中的参数结构必须保存下来；如果必要，被调用的函数有权把寄存器中参数的值存回到内存中（可能是在 C 中，参数是一个指向变量的指针）。

除非调用者确认数据存在浮点寄存器(FP)中更适合，否则，四个寄存器中参数分别存在 a0-a3($4-$7)中。

决定何时以及如何使用 FP 寄存器的标准是很特别的。就风格的 C 没有内建机制检查调用者和被调用者协调函数每个参数的类型。为了帮助程序员避免混乱，调用者将参数转换成固定类型，整数用 int，浮点数用 double。对于分不清整数和浮点数的程序员，实在是没有办法了，不过，这样至少减少了一些混乱。

现代的 C 编译器使用函数原型，定义所有参数类型，并能让所有调用者看到。即使这样，还会有程序的参数类型在编译的时候是不确定的，比如著名的 printf()；printf()是在运行时才确定自己参数的数量和类型。

MIPS 制定了如下的规则。

除非第一个参数是浮点类型，否则不能将后续参数传递到 FP 寄存器中。这样可以保证 printf()等传统函数能正常工作：printf()第一个参数是指针，所以所有后续参数都分配到整数寄存器中，printf()因此能够在参数数据中找到所有参数（不考虑参数类型）。这个规定不会使普通的数学函数效率下降，因为这些函数大多数使用浮点参数。

如果第一个参数是浮点类型，那将会传递到 FP 寄存器中，这样参数结构前 16 字节的其他后续浮点参数也会被传递到 FP 寄存器中。两个 double 占据 16 字节，因此只有两个 FP 寄存器（fa0-fa1 或$f12-$14）用于参数定义。显然不会有人认为函数明确定义大量的单精度参数是常见的事，而非要定义一个新的规则来处理。

另外一个比较特别的是，定义一个函数，它的返回结构大于正常使用的两个寄存器，那么，返回值约定要求产生一个指针作为参数，指向这个结构，并放在其他普通参数前传入（详见 10.7 节）。

如果需要写一个调用约定不是很简单和显而易见的汇编程序，可以先用 C 编写程序，并用编译器带上-S 选项产生汇编文件，以此作为模板。

10.4 C 库范例

这里举个例子：

thesame = strcmp (“bear”, “bearer”, 4);

在 figure 10.1 中，画出了分别参数结构和寄存器内容，这里没有参数数据是放在内存中的，在后续部分会举出那方面的实例。

堆栈位置	内容	寄存器	内容
sp+0	undefined	a0	address of “bear”

sp+4	undefined	a1	address of “bearer”
sp+8	undefined	a2	4
sp+12	undefined	a3	undefined

FIGURE 10.1 **参数结构，三个非浮点操作符**

参数数据不足 16 **字节，所以参数都存在寄存器中。**

看起来，决定将三个参数放在普通寄存器中式荒谬的复杂方法，但是在看看数学库中一些巧妙方法：

```
double  ldexp  ( double ,  int  ) ;
y = ldexp ( x ,  23  ) ; /*  y  =  x  * ( 2  ** 23 ) */
```

Figure 10.2 **显示了相应的参数结构和寄存器值。**

堆栈位置	**内容**	**寄存器**	**内容**
sp+0	undefined	$f12	(double) x
sp+4		$f13	
sp+8	undefined	a2	23
sp+12	undefined	a3	undefined

FIGURE 10.2 **参数传递：一个浮点参数**

10.5 一个特殊的例子：传递数据结构

C **允许使用数据结构类型作为参数（实际上传递的是数据结构的指针，不过** C **语言同时支持这两种方式）。为了适应** MIPS **的规则，传递的数据结构参数只能是参数结构的一部分。在一个** C **的数据结构中，**byte **和** halfword **域会共用一个** word **的位置存放在内存中，因此当通过寄存器传递堆栈中参数结构的参数时，也必须这样处理。**

因此，如果是这样：

```
struct thing {
    char    letter ;
    short   count ;
    int     value ;
```

} = { “z” , 46 , 100000 } ;

(void) processthing (thing) ;

那么， 将会产生 Figure 10.3 **显示的参数结构。**

堆栈位置	内容	寄存器	内容
sp+0	undefined	a0	”z” | x | 46
sp+4	undefined	a1	100000

FIGURE 10.3 **传递数据结构作为参数**

注意，因为 MIPS **的** C **数据结构以域分布的，内存中的顺序和定义数据结构时的顺序是一致的（不过填充时，要尽量满足对齐原则），这些域存放在寄存器中的位置是遵循** load/store **指令的字节顺序，因** CPU **的字节序而异。**Figure 10.3 **是针对大字节序** CPU **的，因此数据结构中的** char **值是在放置参数的寄存器最高** 8 **位，并和** short **对齐。**

如果真想传递数据结构作为参数，而且一定包含短于 short **的数据类型，那就应该测试一下这种情况，看看编译器是否处理正确。**

10.6 传递不定数量的参数

对于传递的参数数量和类型在运行时才能确定的函数，约定对他们的限制是很严格的。考虑这样的例子：

printf (“ length = %f , width = %d \ n”, 1.414 , 1.0 , 12) ;

根据前面的规定，参数结构和寄存器的内容如 Figure 10.4 **所示：**

堆栈位置	内容	寄存器	内容
sp+0	undefined	a0	format pointer
sp+4	undefined	a1	undefined
sp+8	undefined	a2	(double) 1.414
sp+12		a3	
sp+16	(double) 1.0		
sp+20			

sp+24 | undefined |

FIGURE 10.4 printf()参数传递

这里有两件事需要注意。首先，sp+4 中存放的内容需要和 double 类型值对齐（在 C 规则中，浮点参数需要以 double 类型传递，除非通过类型说明和函数原型做出明确的定义）。注意，8 字节内容会导致浪费掉一个标准参数寄存器（译者注：比如 Figure 10.4 中的 a1）。

第二，第一个参数不是浮点参数，根据前述规则，不能给参数分配 FP 寄存器。因此，第二参数的数据（和内存中的一致）只能存放在 a2-a3。

这看似简单，但却是非常实用的。

printf()子程序定义用到的宏是在 stdarg.h 中定义。stdarg.h 提供可移植的接口，用于寄存器和堆栈关于对不定数量和不定类型的操作符进行的操作。printf()解析所有参数，是通过获取第一参数的地址和第二个参数、并在内存中向上寻找参数结构来实现的。

未达到这样的目的，需要 C 编译器将 printf()的 a0-a3 寄存器放到参数结构的范围内。一些编译器会检查是否获得参数的地址，并满足这样的隐性要求；ANSI C 编译器函数定义中使用“…”来做出这样的提示；其他编译器可能使用讨厌的“pragma”来提示，幸运的是，不久 stdarg.h 的宏中就不会有它了。

现在可以看出将 double 类型值放入整数寄存器的必要性了；这样，stdarg 和编译器只要将 a0-a3 存放到参数结构的前 16 字节，而不用考虑参数的数量和类型。

10.7 函数的返回值

函数返回的整数或指针类型，通常是放在寄存器 v0($2)。虽然大多数编译器都不会用到寄存器 v1($3)，不过再 MIPS/SGI 定义的预定中，寄存器 v1($3)是保留不用的。不过在 32 位模式当中返回 64 为非浮点类型的数据时，会用到这个寄存器。有些编译器定义 64 位数据类型(通常是 long long)，这时也会使用 v1 寄存器以返回一个 64 位的数据结构值，而避免 32 位的限制。

所有浮点类型的值都放在寄存器$f0 中返回（在 32 位的 CPU 中，双精度的值也默认使用寄存器$f0）。

如果 C 中的函数返回的数据结构太大，不能通过 v0-v1 返回，就需要作额外处理。这时调用者要在堆栈中为这个数据结构预留一些空间，并用一个指针指向这个空间；被调用的函数将返回值拷贝到这个地方。一般的参数规则要求在函数调用时，将这个指针放在寄存器 a0 传入中，放在 v0 中返回。

10.8 扩展的寄存器使用标准：SGI n32 和 n64

在调用约定和寄存器使用上，n32 和 n64 的 ABI 是一致的（注：不同的是，在 n64 中，long 和指针类型都是 64 位的，而在 n32 中，只有 long long 类型是 64 位的）。

尽管保持寄存器使用约定的一致性时非常有益的事，但是 o32 和 n32/n64 有很大的区别，用不同的方法编译函数，并且不能成功链接到一起。归纳一下 n32/n64 的新规则，主要有以下几点：

- 提供至多 8 个参数通过寄存器来传递。
- 参数和用于参数传递的寄存器都是 64 位的，长度短于整数的类型，都将以 64 位的

形式操作

- 不需要调用者非寄存器中的参数分配堆栈空间
- 尽可能的通过寄存器传递数据结构和数组（传递方法和旧标准相似）
- 前 8 个参数中的浮点值都通过 FP 寄存器传送。实际上，除了在 union 类型和类似 printf()这样参数不确定的函数中，数据结构和数组中和 double 等长的数据类型都将使用 FP 寄存器

允许传递数据结构和数组后，情况变得复杂了，不过即使现在没有堆栈空间可以保留，寄存器的使用相对于复杂参数结构，还是清晰可见的。

n32/n64 废除了 o32 中一个约定，那就是在类似 printf()函数的参数不确定时，要求第一参数必须是非浮点的约定，以便区别普通浮点参数的使用情况。新约定要求调用者和被调用函数在编译时有明确的函数数量和类型，这是通过函数原型来实现的。

n32/n64 有一套不同的寄存器使用约定，Table 10.1 罗列出了于 o32 的区别。唯一的区别在于：单纯的用于临时存储的四个寄存器现在可以传递第 5-8 个参数。对临时寄存器改变（译者注：这里是指 o32 中的 t0-t3 临时寄存器改成 n32/n64 中的 a4-a7 参数寄存器），显然是没有必要的，我对他们的这个做法有点迷惑不解。

TABLE 10.1 新 SGI 工具中整型寄存器使用解决方法

寄存器号	名称	用途
$0	zero	始终为 0
$1	at	汇编编译器使用的临时寄存器
$2,$3	v0,v1	函数返回值
$4-$7	a0-a3	函数参数
$8-$11	t0-t3	临时寄存器
$12-$15	t4-t7	
$24,$25	t8,t9	
$16-$23	s0-s7	保全寄存器（saved register）
$26,$27	k0,k1	留给 interrupt/trap 的处理程序使用
$28	gp	全局指针（Global pointer）
$29	sp	堆栈指针
$30	s8/fp	需要时作为帧指针(Frame pointer)；否则作为附加的保全寄存器
$31	ra	子程序返回的地址

032		n32/n64	
名称	用途	名称	用途
$8-$11	t0-t3	a4-a7	参数
$12-$15	t4-t7	t0-t3	临时寄存器
$24,$25	t8,t9	t8,t9	

表面上，这样可以避免编译器产生的代码丢失曾经存储在四个临时寄存器中的值，但实际上，大多数时候编译器可以使用所有参数寄存器和 v0-v1 寄存器作为临时存储器，达到同样的效果。而且 n32/n64 这一修改，没有影响“保全(saved)”的寄存器(可以假定能从子程序带回值得寄存器)。（注：事实并非如此。在 SGI 计算机上，函数使用 gp 寄存器帮助实现代码的位置独立性，详见 10.11.2。在 o32 中，每个函数以类似的方法使用 gp，这意味着每个函数调用后，不得不恢复 gp 寄存器。在 n32/n64 中，gp 被定义成“保留未使用” 寄存器。

在大多数嵌入式系统中，gp 是常量，因此，上述区别只是理论上的。）

浮点寄存器的约定（见 Table 10.2）是修改比较大的；这没什么可以大惊小怪的，因为 MIPS III CPU 有 16 个额外的 FP 寄存器。在旧的系统体系结构中，偶数寄存器的使用通常会附带使用奇数寄存器（译者注：这样，寄存器的长度就被翻倍使用了，即 2n+0 处的 8 字节内容就放在了 2n+0 和 2n+1 处的两个 4 字节空间里）。（注：MIPS III CPU 有个模式切换，使 FP 的使用和早期的 32 位 CPU 兼容；n32/n64 假定 CPU 不支持这种兼容。）本来 SGI 可以添加一些新的寄存器保持兼容性，但是他们却偏偏要修改大多数现有规则，从新开始。

TABLE 10.2 o32 和 n32/n64 约定中 FP 寄存器使用

寄存器号	在 032 中的用途
$f0,$f2	返回值；fv1 只用于复杂的数据类型，不能在 C 中使用
$f4,$f6,$f8,$f10	临时寄存器，函数中用于不需要保留的值
$f12,$f14	参数寄存器
$f16,$f18	临时寄存器
$f20,$f22, $f24 $f26,$f28, $f30	保全寄存器。为了保证函数调用过程中，这些寄存器中的值长期有效，在对这些寄存器写操作时，必须保存和恢复这些寄存器的值

寄存器号	在 n32 中的用途	在 n64 中的用途
$f0,$f2	返回值；$f2 只用于处理 Fortran 复数返回值正好是两个浮点值，	
$f1,$f3,$f4-$f10	临时寄存器	
$f12-$f19	参数寄存器	
$f20-$f23	偶数($f20-$f30)是临时寄存器	临时寄存器
$f24-$f31	奇数（$f21-$f31）是保全寄存器	保全寄存器

除了可以通过寄存器传递更多参数，n32/n64 没有依赖是否第一参数采取浮点类型而制定新的规则。实际上，参数根据在参数列表中的位置决定如何分配给寄存器。再看看前面的一个例子：

```
double ldexp ( double , int );
y = ldexp ( x , 23 ); /* y = x * (2**23) */
```

Figure 10.5 显示了 n32/n64 中寄存器和堆栈结构的值。

堆栈位置	内容	寄存器	内容
sp+0	undefined	$f12	(double) x
sp+4		$f13	
sp+8	undefined	a1	23

FIGURE 10.5 n32/n64 参数传递：一个浮点参数

尽管 n32/n64 能处理各种浮点和其他类型数据的混合情况，但还是把前 8 个参数中任何

double 类型放在 FP 寄存器中。不过，这不包含 union 类型和参数函数不定的情况（这些情况下，也许不是正真的 double 类型）。注意这是基于函数原型机制，如果没有函数原型机制，那就会有问题。SGI 链接器通常会作检测，并给警告信息。

10.9 堆栈布局、堆栈帧、辅助调试器

Figure 10.6 给出了 MIPS 函数堆栈帧（stack frame）概况图。（现在起，堆栈恢复以前那种增长模式，高内存空间在上面）。传统 MIPS 调用约定要求函数参数前 4 个 word 要给于保留，而在新的调用约定中，只是在需要时分配空间。

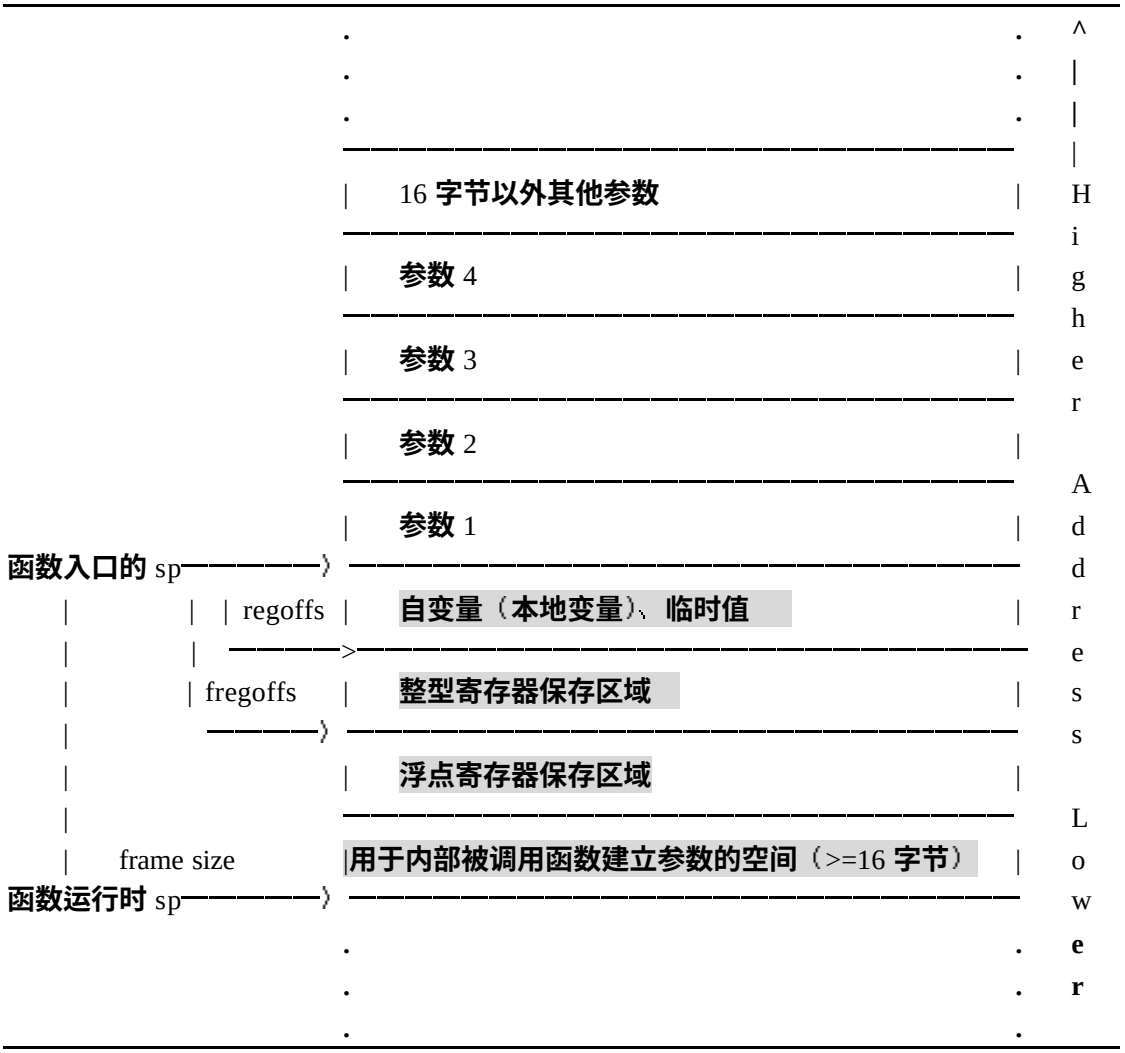


FIGURE 10.6 一个 nonleaf 函数的堆栈帧

（译者注：nonleaf 函数是指内部会调用其他函数的函数，而其他函数就是深层调用 nest call）

显灰的部分是被调用函数自己需要的堆栈空间；上面的白底区域属于调用者。堆栈帧中所有显灰的部分都是可选的，有些函数一个也不需要。非常简单的函数是不需要对堆栈作什么处理的。不过在后续部分，会举例说明其中的一部分。

除了参数（布局需要和调用者一致），堆栈结构对于函数来说，是私有的。需要这种标准布局的唯一理由是用于调试和诊断工具，它们常常需要能够操纵堆栈。如果调试过程中，

中断一个正在运行的函数，通常希望能够在堆栈中回溯，显示运行到当前断点处所调用的函数列表和这些函数的参数列表。而且，希望能够将调试器的上下文回复到堆栈中某个新位置，察看那时某个变量的值；即使维护寄存器中变量数值的代码很少，经过优化的编译器还是能够通过这种机制实现这样的目的。

为了做出正确的分析，调试器必须知道标准的堆栈分布，从而得到必要信息，能够知道每个堆栈帧内容的尺寸和内部布局。如果一个函数在前面的堆栈中用 s0 保存值，以便今后使用，那么调试器需要知道到哪里去找这个保存的值。

在 CISC 体系结构中，经常会有一个复杂的函数调用指令来维系类似于 Figure 10.6 中的堆栈帧，不过还需要一个附加的帧指针寄存器来存储函数入口的 sp。在这样的 CPU 中，调用者的帧指针被保存在大家共知的堆栈位置，以便允许调试器可以忽略堆栈的内容，而只是分析一个简单的链接列表，就能达到目的。但是在 MIPS CPU 中，不需要在运行时间做这些额外工作；大多数时候，编译器知道在函数入口降低堆栈指针和在函数返回时增加堆栈指针。

那么，在这个最小限度的 MIPS 堆栈帧中，调试器如何知道在何处找到曾经保存的值呢？一些调试器做的非常漂亮，甚至通过解析函数前几个指令，就能发现堆栈帧的尺寸和返回地址存放的位置。不过大多数工具根据汇编编译器的指示，在目标代码中适当的放些堆栈帧的信息。

和汇编编译器相关的这种指示很多，因此需要定义一些宏，对这些指示的支持做开关切换，这样免除记忆具体细节差异的痛苦，并在必要时用于和其他工具间的切换。现在大多数工具都已经使用这些宏了；下面的例子使用 LEAF 和 NESTED 宏来实现这样的目的。

对 SGI 约定做全面的描述，是没有什么必要的。下面的例子（使用前面推荐的启始宏和结束宏）是和旧版本 SGI 工具兼容，因此能够被大多数嵌入式工具兼容。

关键的指示.frame 和.mask，可以在 9.5 节找到详细的说明。

为了说明各种问题，我们将函数分成三种类型，并分别加以讲解。

10.9.1 leaf 函数

内部不调用其他函数的函数，成为 leaf 函数。这些函数不用担心建立参数结构和安全维护没有保存的寄存器 t0-t7、a0-a3、v0、v1 数据，可以用堆栈存储数据，并在寄存器 ra 中存放返回地址，在函数运行结束后，通过这个地址直接返回。（注：将返回地址存储在别的地方，可能表现得更好，但是调试器可能找不到。）

有时为了优化或实现 C 无法实现的功能，通常需要用汇编写一些函数，这些函数一般会是 leaf 函数；许多这些函数根本就不使用堆栈空间。声明这样的函数比较简单，比如：

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF    ( myleaf )
------
< your code gose here >
------
j    ra
END     ( myleaf )
```

大多数工具可以在汇编前通过 C 宏预处理器将汇编源码传入，——unix 风格的工具将根据文件的扩展名做出判断。mips/asm.h 和 mips/regdef.h 在定义全局函数和数据的时候非常有用的宏（比如前面的 LEAF 和 END）；同时可以允许直接使用寄存器名字，比如用 a0 来

代表$4。如果使用旧的 MIPS 或 SGI 工具。上面的那段代码将会扩展成:

```
.globl    myleaf
.ent      myleaf,0
------
< your code gose here >
------
j         $31
.end      myleaf
```

其他工具使用的宏可能有所不同，不过实现的功能是大同小异。

10.9.2 nonleaf 函数

内部调用其他函数的函数，称为 nonleaf 函数。通常这些函数需要在开始处，重新设定 sp 寄存器指向所有被其调用函数的参数结构之后的位置，同时保存这些被调用函数所用到的 s0-s8 寄存器最新值。必须保存 ra 寄存器、自变量（也就是堆栈本地变量）和函数在运行后需要保存值的其他寄存器的堆栈位置。（如果参数寄存器 a0-a3 的只需要保存，可以存放到参数结构的标准位置）。

注意，只设定一次 sp（在函数的入口处），所有数据在堆栈中的位置，都通过 sp 加上固定的偏移来获取。

为了解释这点，将通过下面这个 nonleaf 函数说明。（联系 Figure 10.6 的堆栈帧的分布图理解）。

```
#include  <mips/asm.h>
#include       <mips/regdef.h>

#
#myfunc (arg1, arg2, arg3, arg4 ,arg5)
#

#framesize = locals + regsave(ra,s0) + pad  + fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 8 + 4 +8 + (5*4) + 4

NESTED( myfunc , myfunc_frmsz , ra)
     subu          sp,myfunc_frmsz
     .mask    0x80010000 , -4
     sw       ra , myfunc_frmsz-8 (sp)
     sw       s0 , myfunc_frmsz-12 (sp)
     .fmask   0x00300000 , -16
     s.d      $f20 , myfunc_frmsz – 24 (sp)
     ------
     < your code goes here , e.g.>
     # local = otherfunc(arg5,arg2,arg3,arg4,arg1)
     sw       a0 , 16 (sp)                    # arg5(out) = arg1(in)
     lw       a0 , myfunc_frmsz + 16 (sp)     # arg1(out) = arg5(in)
     jal      otherfunc
```

```
sw      v0 , myfunc_frmsz – 4    (sp)
.....
l.d     $f20 , myfunc_frmsz –  24   (sp)
lw      s0 , myfunc_frmsz –  12   (sp)
lw      ra , myfunc_frmsz –  8   (sp)
addu        sp , myfunc_frmsz
jr      ra
END (myfunc)
```

上面代码开始处显示函数 myfunc 有五个参数：在函数入口处，前四个参数放在 a0-a3 中，第五个参数放在 sp+16 处。接下来的这些代码：

```
#framesize = locals + regsave(ra,s0) + pad  + fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 8 + 4 +8 + (5*4) + 4
```

堆栈帧的尺寸计算方式如下：

- local（4 字节）：在堆栈中而不是寄存器保留一个本地变量，可能是需要将这个变量地址传递给其他函数。
- regsave（8 字节）：用来在寄存器 ra 中存放返回地址，因为这个函数会在内部调用其他函数，这可能会用到被调用函数的 s0 保全寄存器
- pad（4 字节）：因为后续的 fregsave 是双精度浮点寄存器，根据规则需要 8 字节对齐，因此这里填补一个 word
- fregsave（8 字节）：用$f20 作为被调用函数的保全浮点寄存器。
- args（20 字节）：内部被调用函数需要五个参数。不过即使这个 nested 函数没有参数，但如果其内部被调用函数是 nested 函数，则这部分的长度不得低于 16 字节。
- pad（4 字节）：堆栈指针也不许是 8 字节对齐的，因此这里也需要一个 word 填补。

接下来的代码：

```
NESTED( myfunc , myfunc_frmsz , ra)
    subu        sp,myfunc_frmsz
```

在 MIPS 公司的工具中，这段代码扩展成：

```
.globl  myfunc
.ent    myfunc , 0
.frame  $29 , myfunc_frmsz , $0
subu    $29 , myfunc_frmsz
```

这里声明了 myfunc 函数的入口，并使它变成全局函数而被访问。.frame 告诉调试器这个函数建立的堆栈的尺寸，subu 指令建立堆栈。

接下来的代码：

```
.mask   0x80010000 , -4
sw      ra , myfunc_frmsz-8 (sp)
sw      s0 , myfunc_frmsz-12 (sp)
```

函数必须保存返回地址和这个堆栈帧中所有被调用函数的保全寄存器。.mask 告诉调试器需要保存的寄存器（$31 和$16，也就是 ra 和 s0）和这些寄存器保存区域顶部相对于堆栈帧顶部的偏移：也就是 Figure 10.6 中的 regoffs。sw 指令保存这些寄存器的值；寄存器号大的存放在堆栈中的位置偏上（也就是寄存器位置排列沿内存地址增加方向的）。接下来的代码：

```
.fmask  0x00300000 , -16
s.d     $f20 , myfunc_frmsz – 24 (sp)
```

对被调函数的保全浮点寄存器$f20 和（隐含的）$f21 做同样的处理。.fmask 的偏移对应于 Figure 10.6 中的 fregoffs（也就是 自变量区域+整形寄存器保存区域+为了对齐而填补的 word）。接下来的代码：

```
# local = otherfunc(arg5,arg2,arg3,arg4,arg1)
sw      a0 , 16 (sp)                    # arg5(out) = arg1(in)
lw      a0 , myfunc_frmsz + 16 (sp)     # arg1(out) = arg5(in)
jal     otherfunc
```

开始调用 otherfunc 函数，这个函数的参数 2-4 是和 myfunc 函数一样的，因此不需要移动，就可以直接传给过来。而参数 1 和参数 5 需要调换一下：将 arg1（在 a0 寄存器中）拷贝到输出参数区域（sp+16）作为被调用函数的 arg5，将 arg5（在 sp+16 处）拷贝到输出参数 1(放在寄存器 a0 中)。

接下来的代码：

```
sw      v0 , myfunc_frmsz – 4   (sp)
```

otherfunc 函数的返回值存放在自变量（本地变量）中；位于堆栈帧的开始 4 字节。

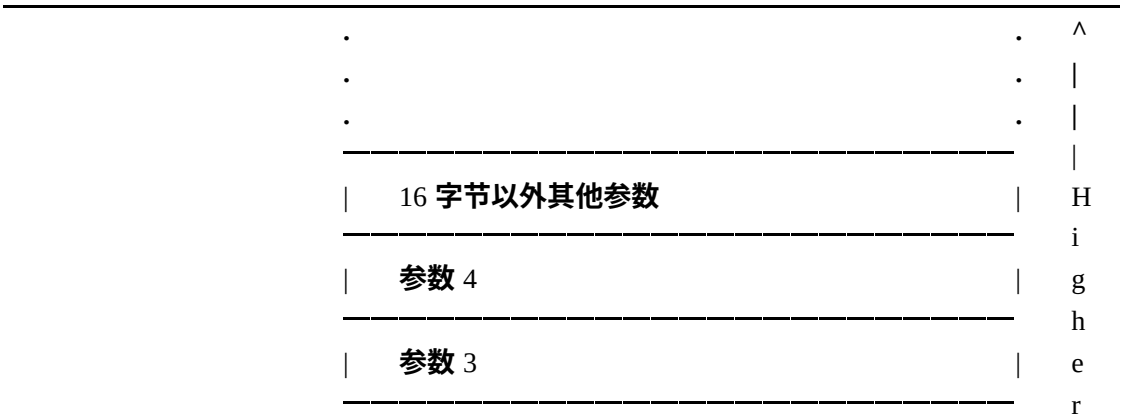
最后：

```
l.d     $f20 , myfunc_frmsz – 24   (sp)
lw      s0 , myfunc_frmsz – 12   (sp)
lw      ra , myfunc_frmsz – 8   (sp)
addu    sp , myfunc_frmsz
jr      ra
END (myfunc)
```

是做一些函数结束时的处理工作：恢复浮点寄存器、整形寄存器和存放返回地址的寄存器；弹出堆栈帧，并返回。

10.9.3 复杂堆栈请求的堆栈帧指针

在前面的堆栈帧描述中，编译器能够管理只需保存 sp 寄存器的堆栈。对于熟悉其他体系结构的程序员来说，经常会使用两个寄存器来管理堆栈，sp 指向堆栈底端，堆栈帧指针指向函数在入口处建立的数据结构。然而，只要编译器能够在函数入口处分配函数所需的堆栈空间，那就能在入口处增加 sp，在函数运行期间，使它指向一个固定的堆栈偏移地址。如果这样，本地堆栈帧中的所有内容，在编译时就确定了相对于 sp 的偏移量，因此不需要额外的堆栈帧指针。但是有时候，在运行过程中堆栈指针会出现混乱：Figure 10.7 显示了 MIPS 是如何针对这种情况分配堆栈帧指针的。



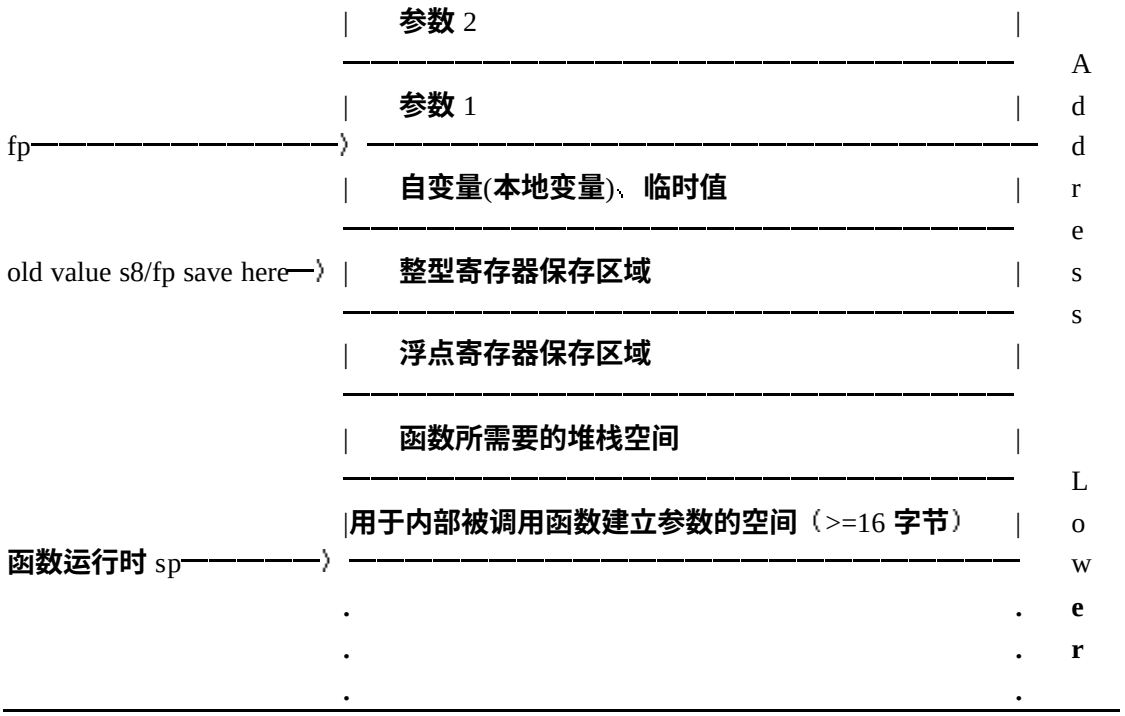


FIGURE 10.7　堆栈帧使用单独的堆栈帧指针寄存器

什么情况下会用到这种机制呢？在一些编程语言中，甚至一些 C 的扩展语言中，会创建在运行时才确定尺寸的动态变量。而且很多 C 编译器使用非常实用的 alloca()内建函数，在运行时按要求分配堆栈空间。这时，函数的入口处，需要使用额外的 s8 寄存器（也就是 fp）来保存此时 sp 值。

既然 fp 寄存器（也就是 s8）是个保全寄存器，函数入口处也必须保存它的原有值，方法和在子程序中作为一个变量保存 s8 一样。在编译时附带了堆栈帧指针的函数里，所有对堆栈帧内部的访问都通过 fp 来获取，因此编译器可以需要降低 sp，为运行时确定尺寸的变量分配空间。

注意，对于内部调用其他函数的函数，并且这个被调用函数使用太多的参数，以至必须使用堆栈传递参数，那么对堆栈帧内部的访问就需要 sp 的帮助了。

这样设计的一个很大好处在于，不管有堆栈帧指针的调用者函数，还是在内部被其调用的函数，都会对这部分进行特殊处理。对于被调用函数来说，因为 fp 寄存器是调用者函数的保全寄存器，因此必须保存 sp 寄存器的值，并在返回是恢复；这样，对于调用者函数，它所看到的堆栈帧中的这部分内容，不会出问题。

汇编工程师很高兴看到，编译器给函数的巨大参数结构保存空间后，还能使通过 alloca()分配空间而返回的地址处于 sp 之上。

有些工具还使用基于 fp 的堆栈帧，以便在本地变量很大而导致一些堆栈帧内容离 sp 太远时，能通过简单的 MIPS load/store 指令（只能访问±32KB 之间的内容）来访问这些内容。

现在再来看看前面一节例子的改进，主要添加了 alloca()的调用：

```
#include      <mips/asm.h>
#include      <mips/regdef.h>

#
#myfunc (arg1, arg2, arg3, arg4 ,arg5)
#
```

```
#framesize = locals + regsave(ra,s8,s0)  +  fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 12  +8 + (5*4) + 4

    .globl    myfunc
    .ent      myfunc , 0
    .frame    fp, myfunc_frmsz, $0

    subu      sp, myfunc_frmsz
    .mask     0xc0010000 , -4
    sw        ra , myfunc_frmsz-8 (sp)
    sw        fp , myfunc_frmsz-12  (sp)   #译者注：fp 就是 s8
    sw        s0 , myfunc_frmsz-16 (sp)
    .fmask    0x00300000 , -16
    s.d       $f20 , myfunc_frmsz – 24 (sp)
    move      fp , sp                      #save bottom of fixed frame
    -----
    # t6 = alloca ( t5 )
    addu      t5 , 7                       #make sure that size
    #and      t5 , ~7                      # is a multiple of 8
    #subu     sp , t5                      # allocate stack
    #addu     t6 , sp , 20                 #leave room for args
    -----
    < your code goes here , e.g.>
    # local = otherfunc(arg5,arg2,arg3,arg4,arg1)
    sw        a0 , 16 (sp)                      # arg5(out) = arg1(in)
    lw        a0 , myfunc_frmsz + 16 ( fp )     # arg1(out) = arg5(in)
    jal       otherfunc
    sw        v0 , myfunc_frmsz – 4   ( fp )    # local = result
    -----
    move      sp , fp                           #restore stack pointer
    l.d       $f20 , myfunc_frmsz – 24    (sp)
    lw        s0 , myfunc_frmsz – 16    (sp)
    lw        fp , myfunc_frmsz – 12    (sp)
    lw        ra , myfunc_frmsz – 8    (sp)
    addu      sp , myfunc_frmsz
    jr        ra
END (myfunc)
```

看看修改过的地方：

```
    .globl    myfunc
    .ent      myfunc , 0
    .frame    fp, myfunc_frmsz, $0
```

这里不再使用 NESTED **宏，是因为使用了独立的堆栈帧指针，而这指针需要通过**.frame **直接明确地进行说明。后面需要修改** fp（**也就是** s8 **或**$30 **寄存器），因此必须在堆栈帧中保存：**

```
.mask    0xc0010000 , -4
sw       ra , myfunc_frmsz-8 (sp)
sw       fp , myfunc_frmsz-12  (sp)
sw       s0 , myfunc_frmsz-16 (sp)
```

接着通过 allocal()在堆栈中分配可变大小(t5B)的空间，并用寄存器 t6 指向这个空间：

```
# t6 = alloca ( t5 )
addu     t5 , 7                    #make sure that size
#and     t5 , ~7                   # is a multiple of 8
#subu    sp , t5                   # allocate stack
#addu    t6 , sp , 20              #leave room for args
```

注意，这里通过处理，将分配空间的尺寸调整到 8 的倍数，以便在满足分配空间要求的同时，使堆栈中内容的地址正确对齐。另外需要注意的是，在堆栈中留出了 20B 空间用于将来调用时存放参数。

在为其他函数建立参数时，使用 sp 寄存器，但是在访问自己的参数和本地变量时，需要使用 fp 寄存器：

```
sw       a0 , 16 (sp)                    # arg5(out) = arg1(in)
lw       a0 , myfunc_frmsz + 16 ( fp )   # arg1(out) = arg5(in)
jal      otherfunc
sw       v0 , myfunc_frmsz – 4  ( fp )   # local = result
```

最后，在函数返回前，恢复堆栈帧指针到在函数入口处所对应的寄存器，并恢复这个寄存器的内容（不要忘记恢复 fp 寄存器的旧值）：

```
move     sp , fp                         #restore stack pointer
l.d      $f20 , myfunc_frmsz – 24   (sp)
lw       s0 , myfunc_frmsz – 16   (sp)
lw       fp , myfunc_frmsz – 12   (sp)
```

10.10 可变长度参数列表

如果要建立一个参数数目不定的函数，需要用到相关工具的 stdarg.h 中定义的一套宏（ANSI 兼容工具必须具备的一套宏）。这套宏定义了 va_start(), va_end()和 va_arg()，具体使用，可以从 Algorithmics 的 SDE-MIPS 中实现的 printf()：

```
int   printf ( const char *format , ··· )
{
      va_start ( arg , format );
      n = vfprintf ( stdout , format , arg );
      va_end (arg );

      return n ;
}
```

一旦调用了 va_start()，就能解析出所有的参数。在给 printf()作参数格式转换的代码中，可以通过下面的方式获得下一个参数，这里假设这个参数是双精度浮点类型：

```
···
d = va_arg( ap , double );
···
```

千万不要在汇编程序中建立参数数目不定的函数，这会引起麻烦。

10.11 不同线程间共享函数和共享库的问题

C 库是包含了一些预编译的模块，在编译时，将被程序所用到的模块中函数和变量，动态链接到程序的二进制代码中。象 printf()一类标准的 C 函数，一般都会包含在 C 库中。

尽管 C 库提供了简单而强有力的方式扩展了 C 语言，但在多任务操作系统中，经常会引起麻烦。通常希望 C 库中的函数能和自己写的代码表现一样——就象是每个任务自己都拥有这部分代码拷贝。但是为了避免对同一内容多次拷贝而大量浪费内存空间，我们总是希望能够最低限度的共享 C 库中的函数代码。库函数很庞大：广泛使用的 X windows 系统的图形接口函数库就会给 MIPS 系统增加 300KB 的内容。

大多数 MIPS 操作系统提供不同任务间共享库代码的机制。为了解释共享函数的问题，下面介绍一下函数会用到的不同数据类型：

- 只读的数据和代码：所有能找到这些数据和代码的线程都可以共享
- 动态数据：特定线程能够安全地保持的参数、函数变量、保全寄存器和函数在堆栈中保持的其他信息。每个任务都要有自己的堆栈空间：即使是与其它线程共享一个地址空间的线程，也必须有自己的堆栈。在函数返回时，这些值都会消失。
- 静态暂时数据：在函数调用期间，不需要保留值的静态数据。原则上，如果不同的几个子函数共享一个静态暂时数据，可以用动态数据来取代它，只不过有点繁琐；这样意味着要重写代码，并重新进行编译。
- 线程范围数据：在库函数调用期间一直存在的静态数据，每个不同线程都要有这个数据的拷贝。全局 errno 变量就是这样的数据（errno 用来处理因调用 UNIX 风格文件 IO 操作函数而引起的出错代码）。
- 全局范围数据：库函数管理的用来跟踪系统状态变化的静态数据。一旦某个库函数为了保持多任务状态的信息而启用了这个数据，它就会成为操作系统的一部分，正常工作；这个内容已经超出了本书的范围。

这些数据类型的说明，在所有线程共用同一地址空间和每个线程使用单独地址空间的情况下，是有很大区别的。

10.11.1 单一地址空间的代码共享

在使用单一地址空间的操作系统，比如大多数实时系统，共享的库函数代码和数据是放在固定的地方；程序寻找库函数和库函数寻找自己的数据，都不会出现问题。然而，库必须可以重入：可供不同的任务使用，而且每个任务都可以在库的某个函数中悬挂，同时这个函数还可以被其他任务使用。动态数据是足够安全的，因此不需要拥有状态信息的简单任务不需要修改，也能工作正常。

静态暂时数据的访问可以通过信号量机制确保数据第一次访问到最后一次访问的连续性（详见 5.8.4）