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。

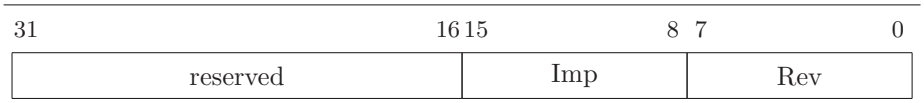

图 3.1: **PRId** 寄存器各个域

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

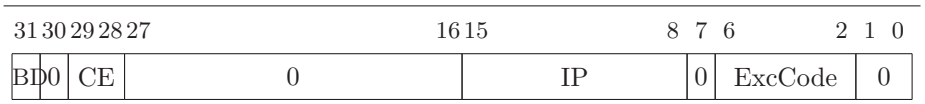

图 3.3: **Cause** 寄存器各个域

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。

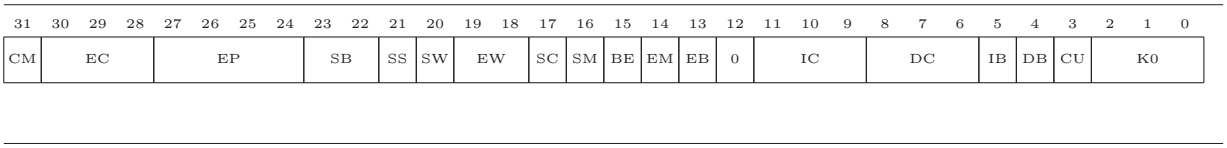

图 3.4: **Config** 寄存器各个域

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。

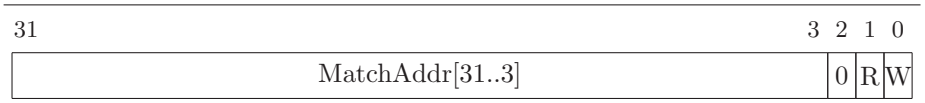

图 3.5: **WatchLo** 寄存器各个域

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。