MIPS CPU 体系结构概述，Linux/MIPS内核

（上）

陈怀临，张福新

弯曲评论、中国科学院计算所

www.tektalk.cn www.ict.ac.cn

前言

2002年，科学院计算所推出基于**MIPS**指令集的龙芯**CPU**。为了推广**MIPS**技术在中国的普及，笔者与计算所龙芯研发组的张福新博士撰写了一系列的关于**MIPS**的技术文章，并被笔者整理发表于笔者的非营利性网站**www.xtrj.org** 上。
六年过去，时光匆匆。笔者这次将其修订并发表于《弯曲评论》以饷读者。希望对读者有所帮助。

第一部分　**MIPS CPU** 体系结构概述

1. MIPS概述

本文介绍**MIPS**体系结构，着重于其寄存器约定，**MMU**及存储管理，异常和中断处理等等。

通过本文，希望能提供一个基本的轮廓概念给对**MIPS CPU**及之上操作系统有兴趣的读者，并能开始阅读更详细的归约**(SPECIFICATION)**资料。

MIPS是最早的，最成功的**RISC(Reduced Instruction Set Computer)**处理 器之一，起源于斯坦福的电机系. 其创始人 **John L. Hennessy**在**1984**年在硅谷创立了**MIPS INC.** 公

司(www.mips.com)。John L. Hennessy目前是Stanford Univ.的校长。在此之前，他是Stanford电子工程学院的院长。计算机专业的学生都知道两本著名的书：
“Computer Organization and Design : The Hardware/Software Interface” 和
“Computer Architecture : A Quantitative Approach”。其作者之一就是**Hennessy**。

MIPS的名字为**“Microcomputer without interlocked pipeline stages”**的缩写。另外一个通常的非正式的说法是**“Millions of instructions per second”**。
2. 指令集

详细的资料请参阅**MIPS**归约。
一般而言，**MIPS**指令系统有：**MIPS I**；**MIPS II**；**MIPS III** 和**MIPS IV**。可想而知，指令系统是向后兼容的。例如，基于**MIPS II**的代码可以在**MIP III**和**MIPS IV**的处理器上运行。

下面是当我们用**gcc**编译器时，如何指定指令和**CPU**的选项。
-mcpu=cpu type
Assume the defaults for the machine type cpu type when scheduling instructions. The choices for cpu type are `r2000', `r3000', `r4000', `r4400', `r4600',
and `r6000'. While picking a specific cpu type will schedule things appropriately for that particular chip, the compiler will not generate any code that does
not meet level 1 of the MIPS ISA (instruction set architecture) without the `-mips2' or `-mips3' switches being used.
-mips1
Issue instructions from level 1 of the MIPS ISA. This is the default. `r3000' is the default cpu type at this ISA level.
-mips2
Issue instructions from level 2 of the MIPS ISA (branch likely, square root instructions). `r6000' is the default cpu type at this ISA level.
-mips3
Issue instructions from level 3 of the MIPS ISA (64 bit instructions). `r4000' is the default cpu type at this ISA level. This option does not change the sizes of any of the C data types.

读者可能发现，对于大多数而言，我们应该是用**MIPS III**或**-mips3**。要提醒的是**R5000**和**R10000**也都是**R4000**的延伸产品。

下面是几点补充：
***MIPS**指令是**32**位长，即使在**64**位的**CPU**上。这对于局部跳转指令的理解很有帮助。

比如：**J (TARGET)**；**JAL (TARGET)**。**J**和**JAL**的**OPERCODE**是**6**位，剩下的**26**为存放跳转

偏移量。由于任何一个指令都是**32**位**(**或**4**字节**)**对齐**(ALIGN)**的，所以**J** 和**JAL**最大的伸缩空间是**2^28=256M**。如果你的程序要作超过**256M**的跳转，你就必须用**JALR**或**JR**，通过一个**GPR**寄存器来存放你的跳转地址。由于一个寄存器是**32**或**64**位的，你就没有任何限制了。

***MIPS CPU**的**SR(STATUS REGISTER)**中有几位是很重要的设置，例如，选择指令系统或要把**64**位的**MIPS**的**CPU CORE**运行在**32**模式下。
SR[XX]：
1：**MIPS IV INSTRUCTION SET USABLE**
0：**MIPS IV INSTRUCTION SET UNUSABLE**

SR[KX]
SR[SX]
SR[UX]：
0：**CPU**工作在**32**位模式下
1：**CPU**工作在**64**位模式下

一般而言，如果你要从头写一个**MIPS**核心为**32**位程序，需要把上述位值设为**0**。

*在以后我们会单独的一章讲将流水线和指令系统，特别是跳转指令的关系。在这里，我们只简单提一下。对任何一个跳传指令后面，要加上一个空转指令**(NOP)**。从而使得**CPU**的流水线不会错误的执行一个预取**(PRE_FETCH)**得指令。当然这个**NOP**可以替换为别的。放一个**NOP**是最简单和安全的。有兴趣的读者可以用**mips64-elf-objdump -d** 来反汇编一个 **OBJECT**文件。就会一目了然了。

*一定要记住：**MIPS I**，**II**， **III**和**IV**指令系统不包含特权指令
换句话说，都是那些在用户态（**USER MODE**）下可以用的指令**(**当然**KERNEL**下也能用**)**。对于**CP0**的操作不属于指令系统。

*有一点在**MIPS CPU**下，要特别注意：对齐（**ALIGN**）。**MIPS**对指令对齐的要求是严厉的。这一点与**POWERPC**是天壤之别。指令必须是**32**位对齐。数据类型必须与她们的大小边界对齐。

*建议读者阅读**MIPS**规约是要花时间看一看指令系统的定义

---

3. 寄存器约定

对于在一个**CPU**上进行开发，掌握其**CPU**的寄存器约定是非常重要的。

MIPS体系结构提供了**32**个**GPR(GENERAL PURPOSE REGISTER)**。这**32**个寄存器的用法大致如下：

REGISTER NAME USAGE
$0 $zero 常量**0(constant value 0)**
$2-$3 $v0-$v1 函数调用返回值**(values for results and expression evaluation)**
$4-$7 $a0-$a3 函数调用参数**(arguments)**

$8-$15 $t0-$t7 暂时的**(**或随便用的**)**
$16-$23 $s0-$s7 保存的**(**或如果用，需要**SAVE/RESTORE**的**)(saved)**
$24-$25 $t8-$t9 暂时的**(**或随便用的**)**
$28 $gp 全局指针**(Global Pointer)**
$29 $sp 堆栈指针**(Stack Pointer)**
$30 $fp 帧指针**(Frame Pointer)**
(fp在现代编译器中基本上不用了，而是作为一个**$t8**来使用。**)**
$31 $ra 返回地址**(Return address)**

对一个**CPU**的寄存器约定的正确用法是非常重要的。当然对**C**语言开发者不需要关心，因为编译器会处理。但对于操作系统核心或驱动程序开发人员就必须清楚。

一般来讲，你通过**objdump -d**可以清醒的看到寄存器的用法。

下面通过笔者写的一个简单例子来讲解：

```
~/ vi Hello.c
"Hello.c" [New file]
/* Example to illustrate mips register convention
* -Author: Huailin Chen
* 11/29/2001
*/

int addFunc(int,int);
int subFunc(int);

void main()
{
```

```
int x,y,z;
x= 1;
y=2;
z = addFunc(x,y);
}


int addFunc(int x,int y)
{
int value1 = 5;
int value2;

value2 = subFunc(value1);
return (x+y+value2);

}

int subFunc(int value)
{
return value--;
}
```

上面是一个**C**程序，**main()**函数调用一个加法的子函数。让我们来看看编译器是如何产生代码的。

```
~/huailinchen:74> /bin/mips-elf-gcc -c Hello.o Hello.c -mips3 -mcpu=r4000 -mgp32 -mfp32 -O1

~/huailinchen:75> /bin/mips64-elf-objdump -d Hello.o
Hello.o: file format elf32-bigmips
Disassembly of section .text:

/* main Function */
0000000000000000 :
/*create a stack frame by moving the stack pointer 8
*bytes down and meantime update the sp value
*/
0: 27bdfff8 addiu $sp,$sp,-8
/* Save the return address to the current sp position.*/
4: afbf0000 sw $ra,0($sp)
```

8: 0c000000 jal 0
/* nop is for the delay slot */
c: 00000000 nop
/* Fill the argument a0 with the value 1 */
10: 24040001 li $a0,1
/* Jump the addFunc */
14: 0c00000a jal 28
/* NOTE HERE: Why we fill the second argument
***behind the addFunc function call?**
*** This is all about the "-O1" compilation optimization.**
*** With mips architecture, the instruction after jump**
*** will also be fetched into the pipline and get**
*** executed. Therefore, we can promise that the**
*** second argument will be filled with the value of**
*** integer 2.**
***/**
18: 24050002 li $a1,2
/*Load the return address from the stack pointer
*** Note here that the result v0 contains the result of**
*** addFunc function call**
***/**
1c: 8fbf0000 lw $ra,0($sp)
/* Return */
20: 03e00008 jr $ra
/* Restore the stack frame */
24: 27bd0008 addiu $sp,$sp,8

/* addFunc Function */
0000000000000028 :
/* Create a stack frame by allocating 16 bytes or 4
*** words size**
***/**
28: 27bdfff0 addiu $sp,$sp,-16
/* Save the return address into the stack with 8 bytes
*** offset. Please note that compiler does not save the**
*** ra to 0($sp).**
***Think of why, in contrast of the previous PowerPC**
*** EABI convention**
***/**

2c: afbf0008 sw $ra,8($sp)
/* We save the s1 reg. value into the stack
*** because we will use s1 in this function**
*** Note that the 4,5,6,7($sp) positions will then**
*** be occupied by this 32 bits size register**
***/**
30: afb10004 sw $s1,4($sp)
/* Withe same reason, save s0 reg. */
34: afb00000 sw $s0,0($sp)
/* Retrieve the argument 0 into s0 reg. */
38: 0080802d move $s0,$a0
/* Retrieve the argument 1 into s1 reg. */
3c: 00a0882d move $s1,$a1
/* Call the subFunc with a0 with 5 */
40: 0c000019 jal 64
/* In the delay slot, we load the 5 into argument a0 reg
***for subFunc call.**
***/**
44: 24040005 li $a0,5
/* s0 = s0+s1; note that s0 and s1 holds the values of
*** x,y, respectively**
***/**
48: 02118021 addu $s0,$s0,$s1
/* v0 = s0+v0; v0 holds the return results of subFunc
***call; And we let v0 hold the final results**
***/**
4c: 02021021 addu $v0,$s0,$v0
/*Retrieve the ra value from stack */
50: 8fbf0008 lw $ra,8($sp)
/*!!!!restore the s1 reg. value */
54: 8fb10004 lw $s1,4($sp)
/*!!!! restore the s0 reg. value */
58: 8fb00000 lw $s0,0($sp)
/* Return back to main func */
5c: 03e00008 jr $ra
/* Update/restore the stack pointer/frame */
60: 27bd0010 addiu $sp,$sp,16

/* subFunc Function */

0000000000000064 :
/* return back to addFunc function */
64: 03e00008 jr $ra
/* Taking advantage of the mips delay slot, filling the
*** result reg v0 by simply assigning the v0 as the value**
***of a0. This is a bug from my c source**
*** codes--"value--". I should write my codes**
*** like "--value", instead.**
68: 0080102d move $v0,$a0

希望读者静下心来把上面的代码看懂。一定要注意编译器为什么在使用**s0**和**s1**之前要先把她们保存起来，然后再恢复，虽然在这个例子中虽然**main** 函数没用**s0**和**s1**。

另外的一点是：由于我们加了**"-O1"**优化，编译器利用了**"delay slot"**来执行那些必须执行的指令，而不是简单的塞一个**"nop"**指令在那里。非常的漂亮。

为了使得读者更加理解寄存器的用法，下面笔者特意提出的一个问题。

*****在写一个核心调度**context switch()**例程时，我们需要**SAVE/RESTORE$t0-$t7**吗？如果不，为什么？

*****在写一个时钟中断处理例程时，我们需要**SAVE/RESTORE$t0-$t7**吗？如果是，为什么？

---

4. MMU和 内存管理

对于**MIPS**的**MMU**和相应的内存管理，读者需要注意的是**:**不存在**x86**或**PowerPC**的实模式。
这一点是**MIPS CPU** 的一个很重要的特点**(**或缺点**)**。

*** MIPS** 存储体系结构

这里笔者不讨论**MIPS64**的存储结构，而只是关注**32**位机器。

MIPS将存储空间划分为**4**大块**--kuseg, kseg0,kseg1 and kseg2.**

0xFFFF FFFF

mapped kseg2
0xC000 0000
unmapped uncached kseg1
0xA000 0000
unmapped cached kseg0
0x8000 0000
2G kuseg
0x0000 0000

对于上述结构，读者要记住以下几点：

***** 当开电**(Power On)**的时候，只有**kseg0 and kseg1** 是可以存取的。

***kseg0 512M(From 0x8000 0000 to 0xA000 0000)**直接被缺省映射到物理内存**0x0000 0000 to 0x2000 0000,** 并且是缓存被开启的（**cache-enabled**）
***kseg1 512M(From 0xA000 0000 to 0xC000 0000)**直接被缺省映射到物理内存**0x0000 0000 t0 0x2000 0000**，并且是没有缓存的（**non cachable**）。
以上两点对于理解基于**MIPS**的驱动程序或操作系统的启动是至关重要的。细心的读者会发现：**kseg1**有点象其他**CPU**的实模式方式。

*****在加电时**(POWER ON)**！**(**虚拟**)**地址**from 0x0000 0000 to 0x8000 0000** 是不可以存取的，必须等到**MMU TLB**初始化之后才可以。

*****同理对从**0xC000 0000** 到**0xFFFF 0000**的地址是不可存取的。

***MIPS**的**CPU**运行有**3**个态**--User Mode**（用户态）**; Supervisor Mode**（管理态） 和**Kernel Mode**（核心态）。 简单而言，读者只需要了解用户态和核心态。操作系统一般而言也只利用这个**CPU**的状态。
下面是读者必须非常清楚的：
×在用户态下，**CPU**能而且只能存取**kuseg**的地址。
×**CPU**必须运行在管理态或核心态下去存取**kseg0**，**kseg1**和**kseg2**的地址空间。
与其他**CPU**一样，**MIPS CPU**是通过**TLB** 来将虚拟地址转换成物理地址。
下面谈谈**MIPS**的**ASID(Address Space Identifier).** 简单而言，**ASID**与虚拟地址一起构成一个定位一个**TLB** 的钥匙（**KEY**）。换句话说，虚拟 地址本身是不能唯一
确定一个**TLB**的。一般而言，在大多数操作系统中，一个**ASID**的值其实就是一个相应的进程标识**ID**。

对于一个多任务操作系统来讲，每个任务都有 自己的**4G**虚拟空间，但是有自己的**ASID**。通过**ASID**，**CPU**可以区分两个具有同样虚拟地址的**TLB**其实是分别属于不同的操作系统的进程或任务。

对**MMU**的处理主要是通过**MMU**的一些控制寄存器来完成的。

MIPS体系结构中集成了一个叫做**System Control Coprocessor (CP0)**的部件。**CP0**就是我们常说的**MMU**控制器。在**CP0**中，除了**TLB**条目**(**例如，对**RM5200**，有**48**对**,96**个**TLB**条目**)**，还提供一系列寄存器提供给操作系统来控制**MMU**的行为。

每个**CP0**控制寄存器都对应一个唯一的寄存器号。**MIPS**提供特殊的指令来对**CP0**进行操作。

mfc0 reg. CP0_REG
mtc0 reg. CP0_REG

通过上述的两条指令来把一个**GPR**寄存器的值赋值给一个**CP0**寄存器，从而达到控制**MMU**的目的。

下面简单介绍几个与**TLB**相关的**CP0**控制寄存器。

Index Register

这个寄存器是用来指定**TLB**条目的，当进行**TLB**读写的时候。例如，**MIPS R5000**提供了**48**个**TLB**对，所以**index**寄存器的值是从**0**到**47**。换句话说，每次**TLB**写的行为是对一对发生的。这一点是与其他的**CPU MMU TLB** 读写不同的。

EntryLo0, EntryLo1

这两个寄存器是用来指定 一个**TLB** 对的偶**(even)**和奇**(odd)**物理**(Physical)**页面地址。

一定要注意的是：**EntryLo0**是给偶数**TLB**页面， **EntryLo1**是给奇数**TLB**页面使用的。否则**MMU**会报异常错误。通常是系统不能启动。

Entry Hi

Entry Hi寄存器存放**VPN2**，或一个**TLB**的虚拟地址部分。注意的是：**ASID** 的值也是在这

里被填写。

Page Mask

MIPS TLB提供可变大小的**TLB**地址映射。一个页面可以
是**4K**，**16K**，**64K**，**256K**，**1M**，**4M**或**16M**。这种可变页大小（**PAGE SIZE**）提供了很好的灵活性，特别是对嵌入式系统软件。 对于嵌入式应用，一个很大的区别就是：不允许大量的页面错处理。否则系统性能 将会非常的差。这一点是传统意义上的操作系统是不一样的。也是为什么**POSIX 1.b**的目的所在。传统**OS**存储管理的一个原则就是：**Page On Demand**。这对大多嵌入式系统是不允许的。 嵌入式系统往往是需要在系统初始化的时刻就对所有的存储进行配置，以确保在系统运行时不会有页面错异常。

上述几个寄存器除了映射一个虚拟页面之外，还包括设置一个页面的属性。其中包括：可写性，合法性，缓存属性等。

下面简单谈谈**MIPS**的**JTLB**。

在**MIPS**中，如**R5000**，**JTLB**的意思是**Joint TLB.** 什么意思呢？就是 **TLB**条目中**TLB**是指令和数据**TLB**混合的。有的**CPU**的指令**TLB**和数**TLB**条目是分开的。

当然**MIPS(R5000)**确实还有两个小的，分开的指令**TLB**和数据**TLB**。但其大小很小。主要是为了提高性能**,**而且是对系统软件透明的。

下面讨论**MMU TLB**和**CPU** 缓存（**Cache**）的关系。

读者知道，**MIPS**，或大多数**CPU**，的**Level 1 Cache**都是采用**Virtually Indexed and Physically Tagged.** 通过这个机制，**OS**不需要在每次进程切换的时候去清除缓存。为什么呢？

举一个例子：

进程**A**的一个虚拟地址**Addr1**，其对应的物理地址是**addre1**；
进程**B**的一个虚拟地址**Addr1**，其对应的物理地址是**addre2;**

在某个时刻，进程**A**在运行中，并且**Addr1**在**Level 1 CACHE**中。
这时候，**OS**进行一个上下文切换，运行进程**B** ，进程**A**进入睡眠状态。
现在，假设进程**B**的第一个指令是虚拟地址**Addr1**进行一个存取。
这时候**CPU**会错误的把进程**A**在缓存中的**Addr1**的**addr1**返回给**CPU**吗？
正确的答案是：不会的。

原因是：
当进程切换时，**OS**会将进程**B**的**ASID**或**PID**填入**ASID**寄存器中。请记住：对**TLB**的访问，**(ASID + VPN)**才是唯一确定**TLB**条目的逻辑。

由于**MIPS**的缓存属性是**Virtually Indexed, Physically tagged.**所以，任何地址的访问，**CPU**都会多**MMU**的**TLB**进行查询，试图找到相应的物理地址。这个物理地址要被用来对**CPU**缓存的条目查找中。**.**

与此同时，**CPU**会把虚拟地址信号传给缓存控制器。然后，我们必须等待上述**MMU**传送过来的物理地址信息。只有物理地址**TAG**也 匹配上了，我们才能说一个：**Cache Hit**（缓存命中）

所以，不需要担心不同的进程有相同的虚拟地址的事情。

---

5. MIPS 异常和中断处理

任何一个**CPU**都要提供一个完善的异常和中断处理机制。一个软件系统，如操作系统，就是一个时序逻辑系统，通过时钟，外部事件来驱动整个预先定义好的逻辑行为。这也是为什么当写一个操作系统时如何定义时间的计算是非常重要的原因。

读者都非常清楚**UNIX**提供了一整套系统调用**(System Call)**。系统调用其实就是一段异常处理程序。

读者可能要问：为什么**CPU**要提供异常和中断处理呢？其目的是：
×处理非法操作。 例如，**TLB Fault**，**Cache Error**等等。
×提供一个通道使得程序可以使用被保护的高级资源，例如**CP0**寄存器。在用户态下的进程不能访问**CP0**。**CPU**通过陷入核心态的异常处理方式使得一个进程可以安全的进行一些**CPU**的高级设置。
×处理外部和内部中断，例如，时钟，看门狗（**WatchDog**）等等。
下面来讨论**MIPS**是如何处理异常和中断。

各种**MIPS**的变种都有些细微的差别。下面是基于**MIPS R7000**的结构。

×理解**MIPS**异常处理最重要的概念是：**MIPS**体系结构采用的是精确异常处理模式。这是什么意思呢？下面来看从**“See MIPS Run”**一书中的摘录：**“In a precise-exception CPU, on any exception we get pointed at one instruction(the exception victim). All instructions preceding the**

exception victim in execution sequence are complete; any work done on the victim and on any subsequent instructions (BNN NOTE: pipeline effects) has no side effects that the software need worry about. The software that handles exceptions can ignore all the timing effects of the CPU's implementations”

上面的意思其实很简单：在发生这个异常之前的一切计算行为会完整的结束并体现效果。在发生这个异常之后的一切计算行为（包含当前这条指令）将不会产生任何效果。

对绝大多数情况而言，如果读者要写一个系统调用**(System Call)**，只需要记住：**MIPS**已经把**syscall**这条指令的地址压在了**EPC**寄存器里。换句话说，在**MIPS**里，你需要在异常返回之前显示的给**EPC**寄存器赋值**EPC<-----EPC+4**。只有这样，你才能从系统调用中正确返回。

下面讨论一下**MIPS**的异常/中断向量表**(Exception/Interrupt Vector)**

MIPS 的异常/中断向量表 **(**假定将**MIPS**运行在**32**为模式下**)**
Reset, NMI： 0x8000 0000
TLB Refill： 0x8000 0000
Cache Error 0xA000 00100
其他的异常都指向：**0x8000 0180**

那么**MIPS**如何来区分和处理一个异常呢？其时序逻辑可以简单的归纳如下：

1. 将**EPC**赋值为要重新执行（被中断的）指令的地址。
2. CPU变为核心态，并且禁止中断。（通过**SR[EXL]**的位。）
3. 对**Cause**寄存器赋值，从而程序可以判断是一个具体的什么异常中断。如果是一个**TLB Miss**方面的异常，**BadVaddre**寄存器也会被同时赋值，从而提高更多的异常信息。
4. CPU开始从异常处理向量的入口存取指令，从而**CPU**逻辑进入异常处理。
5. 从异常返回。

从**MIPS III**指令集开始，从中断返回用的是指令**eret**。 该指令的功能其实就是**iangSR[EXL]**位清零（开中断），并把**CPU**的控制转向将**EPC**寄存器指向的地址。

在理解**MIPS**中断处理结构时，需要对**SR**状态寄存器的**IE**和**EXL**位充分了解。

SR[IE]: 用来开启和关闭中断，其中也包括时钟中断。当然，如果**SR[EXL]**位被置位时，**IE**位不起作用。**EXL**和**ERL**在**CPU**异常中断处理时会被硬件自动置为，从而关闭所有中断，从而系统可以安全的进行中断异常的处理（前期）工作。

K0 and K1 寄存器

这个两个寄存器是被操作系统核心使用的暂时变量，从而不需要使用一些内存变量。读者如果有兴趣的话，可以发现**gcc**编译器的**ABI**不会使用这两个寄存器。换句话说，**K0**和**K1**是系统保留的。
在使用这两个寄存器的时候，要非常小心。例如，当在异常中断处理的中后期，系统软件通常会开启中断，从而系统可以支持嵌套中断处理。这个时候，软件要注意**K0**和**K1**的保存和恢复工作。
笔者不鼓励读者使用**MIPS**的**AT**寄存器。应该使用**.set noat**宏来关闭编译器的优化。
流水线**(Pipeline)** 和中断

读者知道，**MIPS**是一个**RISC**技术处理器。在某一个时刻，在流水线上，同时有若干个指令被处理在不同的阶段**(stage)**上**.**

MIPS处理器一般采用**5**级流水结构。

IF RD ALU MEM WB

那么读者要问：当一个中断异常发生时，**CPU**到底该如何**handle**？答案是这样的：

“On an interrupt in a typical MIPS CPU, the last instruction to be completed before interrupt processing starts will be the one that has just finished its MEM stage when the interrupt is detected. The exception victim will be the one that has just finished its ALU stage...”

对上述的理解是这样的：**CPU** 会保证完成那条已通过 **MEM**流水级的指令。然后将中断牺牲者（**exception victim**）定位在后面那条**(following)**指令上。要注意的是：我们是在谈中断（**Interrupt**）**,** 而不是异常（**Exception**）**.** 在**MIPS**中，这是有细微区别的。

下面介绍几个重要的**SR(Status Register**，状态寄存器**)**与异常和中断有关的位。

*** SR[EXL]**
Exception Level; set by the processor when any exception other than Reset, Soft Reset, NMI, or Cache Error exception are taken. 0: normal 1: exception

当**EXL**被置位时，
- 中断是被禁止的。 换句话说，这时**SR[IE]**位是不管用了，相当于所有的中断都被屏蔽了。

- TLB Refill异常将会使用**General Exception Vector**而不是缺省的**TLB Refill Vector.**
- 如果再次发生异常，**EPC**将不会被自动更新。这一点要非常注意。如果想支持嵌套异常，要在异常处理例程中清**EXL**位。当然要先保存**EPC**的值。另外要注意的：**MIPS**当陷入**Exception/Interrupt**时，并不改变**SR[UX],SR[KX]**或**SR[SX]**的值。**SR[EXL]**为**1**自动的将**CPU mode**运行在核心模式下。这一点要注意。

*** SR[ERL]**
Error Level; set by the processor when Reset, Soft Reset, NMI, or Cache Error exception are taken. 0: normal 1: error
当**ERL**被置位时，
- 中断被禁止。
- 中断返回**ERET**使用的是**ErrorEPC**而不是**EPC**。需要非常注意这个区别。
- Kuseg和**xkuseg** 被认为是没有映射**(Mapped)**的和没有缓存（**Un-Cached**）。可以这样理解，**MIPS CPU**只有在这个时刻才是一种******实模式**(real mode)****，可以不需要**TLB**的映射，就直接使用**kuseg**的地址空间。

*** SR[IE]**
Interrupt Enable 0: disable interrupts 1: enable interrupts。请记住：
当**SR[EXL]**或**SR[ERL]**被**SET**时， **SR[IE]**是无效的。

*** Exception/Interrupt**优先级。

Reset (highest priority)
Soft Reset
Nonmaskable Interrupt (NMI)
Address error --Instruction fetch
TLB refill--Instruction fetch
TLB invalid--Instruction fetch
Cache error --Instruction fetch
Bus error --Instruction fetch
Watch - Instruction Fetch
Integer overflow, Trap, System Call, Breakpoint, Reserved Instruction, Coprocessor Unus-able, or Floating-Point Exception Address error--Data access
TLB refill --Data access
TLB invalid --Data access
TLB modified--Data write
Cache error --Data access
Watch - Data access

Virtual Coherency - Data access
Bus error -- Data access
Interrupt (lowest priority)

读者请注意，所谓的优先级是指：当在某个时刻，同时多个异常或中断出现时，**CPU**将会按照上述的优先级来处理。如果**CPU**目前已经在**TLB refill**处理的例程中，这时，出现了总线错（**Bus Error**）的信号，**CPU**不会拒绝。当然，在这次的处理中，**EPC**的值不会被更新，如果**EXL**还是处于被置位的状态。

异常的嵌套

在有的情况下，希望在异常或中断中，系统可以继续处理其他的异常或中断。
这需要系统软件处理如下事情：

*****进入处理程序后，我们要设置**CPU**模式为核心态，然后清除**SR[EXL],**从而支持**EPC**会被更新，从而支持嵌套处理。

*** EPC** 和**SR**的值

EPC和**SR**寄存器是两个全局的。任何一个异常和中断发生时，**CPU**硬件都会
将更新上述寄存器的当前值。所以，对于支持异常嵌套的系统，要妥善保存**EPC**和**SR**寄存器的值。

***SR[IE]**是一个很重要的位来处理嵌套异常。值得注意的，或容易犯错的一点是：
在做恢复上下文时，要避免重入问题。比如，要用**eret**返回时， 要建立**EPC**的值。在此之前，一定要先关闭中断**disable interrupt.** 否 则，**EPC**可能被冲掉。

下面是一段异常中断返回的例子代码：

```
/* 读取SR的当前值*/
mfc0 t0,C0[SR]
/*加一个delay slot指令 */
nop
/* 清楚SR[IE]，关闭中断 */
li t1,~SR[IE]
and t0,t0,t1
mtc0 t0,C0[SR]
```

nop
/* 可以安全的恢复**EPC**的值***/**
ld t1,R_EPC(sp)
mtc0 t1,C0[EPC]
nop
lhu k1, /* 恢复老的中断屏蔽码，被暂时保留在**k1**里***/**

or t0,t0,k1

/*从新对**SR[EXL]**置位。**ERET**会自动将其清除。一定要理解 ，为什么中断例程要在前面要清除 **EXL**。如果不的话。就不能支持嵌套异常。为什么，希望读者能思考并回答。并且，在清**EXL**之前，我们一定要先把**CPU**模式变为核心模式。***/**
ori t0,t0,SR[EXL]
/*一切就绪，恢复中断屏蔽码和对**EXL**置位***/**
mtc0 t0,C0[SR]
nop
ori t0,t0,SR[IE]
/* 置为**IE */**
ori t0,t0,SR[IMASK7]
mtc0 t0,C0[SR]
nop
/*恢复**CPU**模式 ***/**
ori t0, t0,SR[USERMODE]
mtc0, t0, C0[SR]

eret
/*eret将对**EXL**清零。所以要注意，如果你在处理程序中改变了**CPU**的模式，一定要确保，在重新设置**EXL**位后，恢复**CPU**的原来模式，否则用户进程将会在核心态下运行。