MIPS CPU 体系结构概述　http://www.xtrj.org/mips/

陈怀临

1。序言

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OS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Stanford Univ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Dean。CS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Co-author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".

MIPS 芯片在工业界目前用的比较多的是：MIPS INC。的R10000；
QED(http://www.qedinc.com。1996年从MIPS INC。分(SPIN OFF)出来的)的R5000，R7000等。

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的处理器上跑一跑：-)

下面是当我们用gcc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，当我们选择指令系统或要用64位的MIPS的CPU CORE在32模式下(绝大多数情况，弟兄们 别告诉我你在写64位的程序：--))。

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，最好把上述值设为 0。为什么最好呢？因为我在工作中没有去冒风险，设她们为 1，who knows what would happen?:-) And then why bother:--)？

*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，FOR SIMPLITY，要加上一个空转指令(NOP)。从而使得 CPU 的 PIPELINE 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。以后再讲。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。你就会一目了然了。

*一定要记住：MIPS I，II， III 和 IV 指令系统不包含 PRIVILEDGED INSTRUCTIONS。

换句话说，都是那些在 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，要千万注意：ALIGN。MIPS 对 ALIGN 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。数据类型必须在她们的的大小边界对齐。简单的比如：When CPU running under 32bit mode, int must 32bit aligned; long 32bit aligned; pointer must be 32bit aligned; char must 8 bit aligned. long long must 64 bit aligned;关于这一点 ，我是吃过苦头的。当然我知道大家还会犯错在这里：--)，即使知道了。有些事情学是没用的：--)。一定要注意。

*我建议读者阅读 SPECIFICATION 时要花时间看一看指令系统的定义。其实不难。每一种指令不外乎几个域(FIELDS)。

寄存器约定

对于在一个 CPU 上进行开发，掌握其工作的 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)
(BNN：fp is stale acutally, and can be simply used as $t8)
$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COMPILER会TAKE CARE。但对于KERNEL的开发或DRIVER开发的人就**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。

下面通过我刚才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: BNN
* 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。

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

~/bnn: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 optimizaiton.
* With mips architecture, the instruciton after jump
* will also be fetched into the pipline and get
* exectuted. 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
```

希望大家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 SAVE 起来，然后再 RESTORE，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。

最后，考大家一个问题，为了使得大家更加理解寄存器的用法：

*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？

MMU和 Memory Management

对于MIPS的MMU和Memory Management, the first and yet important one we need
always
keep in mind is: No real-mode

没有实模式。这一点是MIPS CPU 的一个很重要的特点(或缺点)。

我们会问了：BNN，Give me a break. Without CPU running in the real-mode, how could you boot up a kernel? Well, here is the thing:

Bydefault, MIPS architecture , when power on, has enabled/mapped two memory
areas. In other words, those two memory areas are the places where your boot codes HAVE TO resident and run on top of. If you read the makefiles

of MIPS linux source tree, you would easily find the infor. For example,

0x8000xxxx or some things like that.

* MIPS 存储体系结构

我们在这里不谈64位CPU，只谈32位的。

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

对于上述图表，弟兄们要记住以下几点：

* 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) are DIRECTLY mapped to phyiscalμ
memory ranging from 0x0000 0000 to 0x2000 0000, with cache-able(either write
back or write through, which is decided by SR(Status Register of MIPS CPU)μ

*kseg1 512M(From 0xA000 0000 to 0xC000 0000) are (also) DIRECTLy mapped to physical memory ranging from 0x0000 0000 t0 0x2000 0000, with non-cachable.

以上两点对于理解MIPS OS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real-mode方式。

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

*同理对地址from 0xC000 0000 to 0xFFFF 0000

*MIPS的CPU运行有3个态--User Mode; Supervisor Mode and Kernel Mode. For simplicity, let's just talk about User Mode and Kernel Mode. Please always keep this in mind:

CPU can ONLY access kuseg memory area when running in User Mode
CPU MUST be in kernel mode or supervisor mode when visiting kseg0, kseg1

and kseg2 memory area.

* MMU TLB

MIPS CPU通过TLB 来translates all virtual addresses generated by the CPU.对
于这一点，这里不多废话。

下面谈谈ASID(Address Space Identifier). Basically, ASID, plus the VA(Virtual
Address) are composed of the primary key of an TLB entry. 换句话说，虚拟

地址本身是不能唯一
确定一个TLB entry的。一般而言，ASID的值就是相应的process ID.

Note that ASID can minimized TLB re-loads, since several TLB entries can

have the same virtual page number, but different ASID's. 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。

MMU 控制寄存器

对于一个Kernel Engineer来说，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 entry(例如，对RM5200，有48pair,96个TLB entry),一些控制寄存器提供给OS KERNEL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

我们通过上述的两条指令来把一个GPR寄存器的值assign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 entry的，当你进行TLB读写的时候。我们已经知道，例如，MIPS R5提供了48个TLB pair，所以index寄存器的值是从0到47。换句话说，每次TLB写的行为是对一个pair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specify 一个TLB pair的偶(even)和奇(odd)物理(Physical)页面地址。

一定要注意的是：EntryLo0 is used for even pages; EntryLo1 is used for odd pages.
Otherwise, the MMU will get exception fault.

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 value 也是在这里被体现。

Page Mask

MIPS TLB 提供可变大小的 TLB 地址映射。一个 PAGE 可以是
4K，16K，64K，256K，1M，4M 或 16M。这种可变 PAGE SIZE 提供了很好的灵活性，特别是对 Embedded System Software. 对于 Embedded System Softare,一个很大的区别就是：不允许大量的 Page Fault.
这一点是传统 OS 或 General OS 在 Embedded OS 上的致命缺陷。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 Embedded System 是不允许的。 For embedded system,往往是需要在系统初始化的时刻就对所有的
存储进行 configuration， 以确保在系统运行时不会有 Page Fault.

上述几个寄存器除了 MAP 一个虚拟页面之外，还包括设置一个页面的属性。其中包括

writable or not; invalide or not; cache write back or write through

下面简单谈谈 MIPS 的 JTLB。

在 MIPS 中，如 R5000， JTLB is provided. JTLB stands for Joint TLB. 什么意思呢？就是
TLB buffer 中包含的 mixed Instruction and Data TLB 映射。有的 CPU 的 Instruction TLB 和 Data TLB buffer 是分开的。

当然 MIPS(R5000)确实还有两个小的，分开的 Instruction TLB 和 Data TLB。但其大小很小。主要是为了 Performance,而且是对系统软件透明的。

在这里再谈谈 MMU TLB 和 CPU Level 1 Cache 的关系。

我们知道，MIPS，或大多数 CPU，的 Level 1 Cache 都是采用 Virtually Indexed and Physicall tagged. 通过这个机制，OS 就不需要在每次进程切换的时候去 flush CACHE。为什么呢？

举一个例子吧：

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 does a context swith and bring process B up, having process A sleep.

Now, let's assume that the first instruction/data fetch process B does is to access its own virtual address Addr1.

这时候CPU会错误的把进程A在Level 1中的Addr1的addr1返回给CPU吗?

我们的回答应该是：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Primary Key.

由于MIPS的CACHE属性是Virtually Indexed, Physically tagged.所以，任何地址的访问，CPU都会issue the request to MMU for TLB translation to getμ the correct physical address, which then will be used for level cache matching.

与此同时，CPU会把虚拟地址信号传给Level 1 Cache 控制器。然后，我们必须等待MMU的Physical Address数据。只有physical tag也 匹配上了，我们才能说一个：Cache Hit.

所以，我们不需要担心不同的进程有相同的虚拟地址的事情。

弟兄们可以重温一下我们讲过的Direct Mapped; Full Associative, and Set Associative.

从而理解为什么Cache中可以存在多个具有相同虚拟地址的entry. For example,the above Addr1 for proccess A and Addr1 for process B.

MIPS 异常和中断处理(Exception and Interrupt handling)

任何一个CPU都要提供一个详细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。

大家都非常清楚UNIX提供了一整套系统调用(System Call)。系统调用其实就是一段EXCEPTION处理程序。

我们可能要问：为什么CPU要提供Excpetion 和 Interrupt Handling呢?

*处理illegal behavior, 例如，TLB Fault, or, we say, the Page fault; Cache Error;

* Provide an approach for accessing priviledged resources, for example, CP0 registers. As we know, for user level tasks/processes, they are running
with the User Mode priviledge and are prohibilited to directly control CP0. CPU need provide a mechanism for them to trap to kernel mode and then safely manipulate resources that are only available
when CPU runs in kernel mode.

* Provide handling for external/internal interrupts. For instance, the timer interrupts and watch dog exceptions. Those two interrupt/exceptions are very important for an embedded system applicances.

Now let's get back to how MIPS supports its exception and interrupt handling.

For simplicty, all information below will be based on R7K CPU, which is derived from the R4k family.

* The first thing for understanding MIPS exception handling is: MIPS adopts **Precise Exceptions** mechanisms. What that means? Here is the explaination from the book of "See MIPS Run": "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EXCEPTION之前的一切计算行为会**FINISH**。在发生EXCEPTION之后的一切计算行为将不需考虑。

对绝大多数情况而言，如你要写一个系统调用(System Call),你只要记住：
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，compard to the PowerPC CPU srr1 register, 你需要**explicitely** refill the EPC register by EPC<-----EPC+4, before you use the eret 中断返回。只有这样，你才能从系统调用中正确返回。

异常/中断向量(Exception/Interrupt Vector)

MIPS 的 Exception/Interrupt Vector 的 organizaion is not as good as PowerPC CPUs.

For PPC, every detailed exception cause is directed to a unqiue vector address. MIPS is otherwise. Below is a recap of MIPS exception/interrupt vectors.
(We herein only talk about running MIPS CPU in the 32 bit mode)
Reset, NMI 0x8000 0000
TLB refill 0x8000 0000
Cache Error 0xA000 00100 (BNN: Why goes to 0xAxxxxx? A question to readers. Please think about the difference between Kseg0 and kseg1)
All other exceptions 0x8000 0180

How MIPS acts when taking an exception?

1. It sets up the EPC to point to the restart location.
2. CPU changes into kernel mode and disables the interrupts (BNN: MIPS does this by setting EXL bit of SR register)
3. Set up the Cause register to indicate which is wrong. So that software can tell the reason for the exception. If it is for address exception, for example, TLB miss and so on, the BadVaddr register is also set.
4. CPU starts fetching instructions from the exception entry point and then goes to the exception handler.

Returning from exceptions

Up to MIPS III, we use the eret instruciton to return to the original location before falling into the exception. Note that eret behavior is: clear the SR[EXL] bit and returns control to the adress stored in EPC.

An important bit in SR for interrupt handling

SR[IE]: This bit is used to enable/disable interrupts,, including the

timer interrupts. Of couse,
when the SR[EXL] bit is set, this bit has no effects.

K0 and K1 registers:

These two registers are mostly used by kernel as a temporary buffer to hold some values if necessary. So that you don't have to find some pre-defined
memories for that purpose.

One thing we should be careful is : When you are allowing the nested exception/interrupt handling, you need take care of these two registers' values as
they will be over-written, for example.

I don't encouarge people to use the AT register too often, even though you can use the .set noat directive. I have found a bug in mips-gcc, which will use the AT register anyway, even after we use the .set noat. In other wrods, using AT is dangeous somhow if you are not quite familire with the register convention/usage

流水线(Pipeline) and Interrupt Taken

我们知道，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

那么我们要问：当一个 Interrupt 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**完成**那条已**finish** MEM stage 的指令。然后将 exception victim 定位在下一条(following)指令上。要注意的是：我们是在谈 Interrupt, not the exception. 在 MIPS 中，这是有区别的。

下面介绍几个重要的 SR(Status Register)与 Exception 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

When EXL is set:
- Interrupts are disabled. 换句话说，这时SR[IE]位是不管用了，相当于所有的中断都被MASK了。
- TLB refill exceptions will use the general exception vector instead of the TLB refill vector.
- EPC is not updated if another exception is taken. 这一点要注意。如果我们想支持nesting exceptions, 我们要在exception hander中clear EXL bit.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KERNEL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
When ERL is set:
- Interrupts are disabled.
- The ERET instruction uses the return address held in ErrorEPC instead of EPC.
- Kuseg and xkuseg are treated as unmapped and uncached regions.This allows main memory to be accessed in the presence of cache errors. 这时刻，我们可以说，MIPS CPU只有在这个时刻才是一种**实模式(real mode)**.

* 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)

大家请注意，所谓的优先级是指：当在某个时刻，同时多个Exception或Interrupt出现时，CPU将会按照上述的优先级来TAKE。如果CPU目前在，for example,TLB refill处理handler中，这时，出现了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是SET的话。

Nesting Exceptions

在有的情况下，我们希望在Exception或中断中，系统可以继续接送exception或中断。
这需要我们小心如下事情：

*进入处理程序后，我们要设置CPU模式为KERNEL MODE然后重新clear SR[EXL], 从而支持EPC会被更新，如果出现新的Exception/Interrupt的话。

* EPC 和SR的值

EPC和SR寄存器是两个全局的。任何一个Exception/Interrupt发生时，CPU硬件都会
将其value over-write.所以，对于支持Nesting Exceptoins的系统，要妥善保存EPC和SR寄
存器的VALUE。

* EPC 和SR的值 EPC和SR寄存器是两个全局的。任何一个Exception/Interrupt发生时，CPU硬件都会 将其value over-write.所以，对于支持Nesting Exceptoins的系统，要妥善保存EPC和SR寄 存器的VALUE。SR[IE]是一个很重要的BIT来处理在处理Nesting Exception时，值得注意的，或容易犯错的一点是(我在这上面吃过苦头)：
一定要注意：在做 restore context时，要避免重入问题。比如，但要用eret返回时， 我们要set up the EPC value. 在此之前，一定要先disable interrupt. 否

则，EPC value 可能被冲掉。

下面是一段codes of mine for illustrating the exception return.

```
restore_context
/* Retrieve the SR value */
mfc0 t0,C0_SR
/* Fill in a delay slot instruction */
nop
/* Clear the SR[IE] to disable any interrupts */
li t1,~SR_IE
and t0,t0,t1
mtc0 t0,C0_SR
nop
/* We can then safely restore the EPC value * from the stack */
ld t1,R_EPC(sp)
mtc0 t1,C0_EPC
nop
lhu k1, /* restore old interrupt imask */
or t0,t0,k1
/* We reset the EXL bit before returning from the exception/interrupt
the eret instruction will automatically clear the EXL then. 一定要理解
我为什么要在前面clear EXL.如果不得话。就不能支持nesting exceptions. 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
KERNEL MODE。
*/
ori t0,t0,SR_EXL
/* restore mask and exl bit */
mtc0 t0,C0_SR
nop
ori t0,t0,SR_IE
/* re-set ie bit */
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ATOMIC的将EXL清零。所以要注意，如果你在处理程序中改变了CPU
得模式，例如，一定要确保，在重新设置EXL位后，恢复CPU的original mode.
Otherwise,for example, a task/process will run in kernel mode. That
would be totally mess up your system software.*/
```

In summary, exception/interrupt handling is very critical for any os kernel. For a kernel engineer, you should be very clear with the exception mechanisms of your target CPU provides. Otherwise, it would cost you bunches of time for bug fixes.

Again, the best way is to read the CPU specification slowly and clearly. There is no any better approach there. No genius, but hard worker, always.

Mips kernel Introduction

1.硬件知识
 * CPU 手册: http://www.mips.com 等.
 * 主板资料,找你的卖家.
 * 背景知识:如 PCI 协议,中断概念等.

2.软件资源

 * http://oss.sgi.com/linux,ftp://oss.sgi.com
 * http://www.mips.com
 * mailing lists:
    linux-mips@oss.sgi.com
    debian-mips@oss.sgi.com
 * kernel cvs
   sgi:
     cvs -d :pserver:cvs@oss.sgi.com:/cvs login
    (Only needed the first time you use anonymous CVS, the password is "cvs")
     cvs -d :pserver:cvs@oss.sgi.com:/cvs co linux
   另外 sourceforge.net 也有另一个内核树, 似乎不如 sgi 的版本有影响.
 * 经典书籍:
      * "Mips R4000 Microprocessor User's Manual",by Joe Heinrich
      * "See Mips Run",by Dominic Sweetman
 * Jun Sun's mips porting guide: http://linux.junsun.net/porting-howto/
 * 交叉编译指南:http://www.ltc.com/~brad/mips/mips-cross-toolchain.html
 * Debian Mips port: http://www.debian.org/ports/mips
 * 系统杂志网站: http://www.xtrj.org

3. mips kernel 的一般介绍

(下面一些具体代码基于2.4.8的内核)
我们来跟随内核启动运行的过程看看mips内核有什么特别之处.

加电后,mips kernel从系统固件程序(类似bios,可能烧在eprom,flash中)得到控制
之后(head.S),初始化内核栈,调用init_arch初始化硬件平台相关的代码.

init_arch(setup.c)首先监测使用的CPU(通过MIPS CPU的CP0控制寄存器PRID)确定使用的指令集和一些CPU参数,如TLB大小等.然后调用prom_init做一些底层参数初始化. prom_init是和具体的硬件相关的.

使用MIPS CPU的平台多如牛毛, 所以大家在arch/mips下面可以看到很多的子目录,每个子目录是一个或者一系列相似的平台.这里的平台差不多可以理解成一块主板加上它的系统固件,其中很多还包括一些专用的显卡什么的硬件(比如一些工作站).
这些目录的主要任务是:

1. 提供底层板子上的一些重要信息,包括系统固件传递的参数,io的映射基地址,内存的大小的分布等.多数还包括提供早期的信息输入输出接口(通常是一个简单的串口驱动)以方便调试,因为pmon往往不提供键盘和显示卡的支持.?
2. 底层中断代码,包括中断控制器编程和中断的分派,应答等
3. pci子系统底层代码. 实现pci配置空间的读写,以及pci设备的中断,IO/Mem空间的分配
4. 其它,特定的硬件.常见的有实时时钟等

这里关键是要理解这些硬件平台和熟悉的x86不同之处.我印象较深的有几个:

* item MIPS不象X86有很标准的硬件软件接口,而是五花八门,每个厂家有一套,因为它们很多是嵌入式系统或者专门的工作站.不象PC中,有了BIOS后用同一套的程序,就可以使用很多不同的主板和CPU.

  MIPS中的'bios'常用的有pmon和yamon,都是开放源代码的软件。很多开发板带的固件功能和PC BIOS很不一样,它们多数支持串口显示,或者网络下载和启动,以及类DEBUG的调试界面,但可能根本不支持显卡和硬盘,没有一般的基本'输入输出'功能.
* PCI系统和地址空间,总线等问题.
  在x86中,IO空间用专门的指令访问,而PCI设备的内存空间和物理内存空间是相同的,也就是说,在CPU看来物理内存从地址0开始的话,在PCI设备看来也是一样的.反之,PCI设备的基地址寄存器设定的PCI存储地址,CPU用相同的物理地址访问就行了.

  而在MIPS中就很不一样了,IO一般是memory map的,map到哪里就倚赖具体平台了.而PCI设备的地址空间和CPU所见的物理内存地址空间往往也不一样(bus address & physical address).所以mips kernel的iob/outb,以及

bus_to_virt/virt_to_bus,phys_to_virt/virt_to_phys,ioremap 等就要小心考虑.这些问题有时间我会对这些问题做专门的说明.

PCI 配置空间的读写和地址空间映射的处理通常都是每个平台不一样的.
因为缺乏统一接口的 BIOS,内核经常要自己做 PCI 设备的枚举,空间分配,中断分配.

* 中断系统.
  PC 中中断控制器先是有 8259,后来是 apic,而 cpu 的中断处理 386 之后好像也变化不大,相对统一.
  mips CPU 的中断处理方式倒是比较一致,但是主板上的控制器就乱七八糟了怎么鉴别中断源,怎么编程控制器等任务就得各自实现了.
  总的说来,MIPS CPU 的中断处理方式体现了 RISC 的特点:软件做事多,硬件尽量精简. 编程控制器,提供中断控制接口,dispatch 中系?这一部分原来很混乱,大家各写各的,现在有人试图写一些比较统一的代码(实际上就是原来 x86 上用的 controller/handler 抽象).

* 存储管理.
  MIPS 是典型的 RISC 结构,它的存储管理单元做的事情比象 x86 这种机器少得多.例如,它的 tlb 是软件管理的,cache 常常是需要系统程序干预的.而且,过多的 CPU 和主板变种使得这一部分非常复杂,容易出错.存储管理的代码主要在 include/
  asm-mips 和 arch/mips/mm/目录下.

* 其它.
  如时间处理,r4k 以上的 MIPS CPU 提供 count/compare 寄存器,每隔几拍 count 增加, 到和 compare 相等时发生时钟中断,这可以用来提供系统的时钟中断.但很多板子
  自己也提供其它的可编程时钟源.具体用什么就取决于开发者了.

init_arch 后是 loadmmu,初始化 cache/tlb.代码在 arch/mips/mm 里.有人可能会问,
在 cache 和 tlb 之前 CPU 怎么工作的?
在 x86 里有实模式,而 MIPS 没有,但它的地址空间是特殊的,分成几个不同的区域,每个区域中的地址在 CPU 里的待遇是不一样的,系统刚上电时 CPU 从地址 bfc00000 开始,那里的地址既不用 tlb 也不用 cache,所以 CPU 能工作而不管 cache 和 tlb 是什么
样子.当然,这样子效率是很低的,所以 CPU 很快就开始进行 loadmmu. 因为 MIPS CPU
变种繁多,所以代码又臭又长. 主要不外是检测 cache 大小,选择相应的 cache/tlb flush 过程,还有一些 memcpy/memset 等的高效实现.这里还很容易出微妙的错误,软件管理 tlb 或者 cache 都不简单,要保证效率又要保证正确.在开发初期常常先关掉 CPU 的 cache 以便排除 cache 问题.

MMU 初始化后,系统就直接跳转到 init/main.c 中的 start_kernel,很快吧?

不过别高兴,start_kernel 虚晃一枪,又回到 arch/mips/kernel/setup.c,调用 setup_arch,这回就是完成上面说的各平台相关的初始化了.

平台相关的初始化完成之后,mips 内核和其它平台的内核区别就不大了,但也还有不少问题需要关注.如许多驱动程序可能因为倚赖 x86 的特殊属性(如 IO 端口,自动的 cache 一致性维护,显卡初始化等)而不能直接在 MIPS 下工作.

例如,能直接(用现有的内核驱动)在 MIPS 下工作的网卡不是很多,我知道的有 intel eepro100,AMD pcnet32 ,Tulip. 3com 的网卡好像大多不能用.显卡则由于 vga bios 的问题,很少能直接使用.(常见的显卡都是为 x86 做的,它们常常带着一块 rom,里面含有 vga bios,PC 的 BIOS 的初始化过程中发现它们的化就会先去执行它们以初始化显卡,然后才能很早地在屏幕上输出信息).而 vga bios 里面的代码一般是 for x86,不能直接在 mips CPU 上运行.而且这些代码里常常有一些厂家相关的特定初始化,没有一个通用的代码可以替换.只有少数比较开放的厂家提供足够的资料使得内核开发人员能够跳过 vga bios 的执行直接初始化他的显卡,如 matrox.

除此之外,也可能有其它的内核代码由于种种原因(不对齐访问,unsigned/signed 等)不能使用,如一些文件系统(xfs?).

关于 linux-mips 内核的问题,在 sgi 的 mailing list 搜索或者提问比较有希望获得解决.如果你足够有钱,可以购买 montivista 的服务.http://www.mvista.com.

4.mips 的异常处理

1.硬件

mips CPU 的异常处理中,硬件做的事情很少,这也是 RISC 的特点. 和 x86 系统相比,有两点大不一样:

* 硬件不负责具体鉴别异常,CPU 响应异常之后需要根据状态寄存器等来确定究竟发生哪个异常.有时候硬件会做一点简单分类,CPU 能直接到某一类异常的处理入口.
* 硬件通常不负责保存上下文.例如系统调用,所有的寄存器内容都要由软件进行必要的保存.

各种主板的中断控制种类很多,需要根据中断控制器和连线情况来编程.

2.kernel 实现

* 处理程序什么时候安装?

traps_init(arch/mips/kernel/traps.c,setup_arch 之后 start_kernel 调用)

```
    ...
    /* Copy the generic exception handler code to it's final
destination. */
```

```
    memcpy((void *)(KSEG0 + 0x80), &except_vec1_generic, 0x80);
    memcpy((void *)(KSEG0 + 0x100), &except_vec2_generic, 0x80);
    memcpy((void *)(KSEG0 + 0x180), &except_vec3_generic, 0x80);
    flush_icache_range(KSEG0 + 0x80, KSEG0 + 0x200);
    /*
     * Setup default vectors
     */
    for (i = 0; i <= 31; i++)
    set_except_vector(i, handle_reserved);
    ...
```

* 装的什么?

except_vec3_generic(head.S) (除了TLB refill例外都用这个入口):

```
  /* General exception vector R4000 version. */
NESTED(except_vec3_r4000, 0, sp)
.set    noat
mfc0    k1, CP_CAUSE
andi    k1, k1, 0x7c  /* 从cause寄存器取出异常号 */
li      k0, 31<<2      beq     k1, k0, handle_vced /* 如果是vced,
处理之*/      li     k0, 14><<2      beq     k1, k0, handle_vcei /* 如
果是vcei,处理之*/      /* 这两个异常是和cache相关的,cache出了问题,不能再在
这个cached的位置处理啦 */      la     k0, exception_handlers /* 取出异常
处理程序表 */      addu    k0, k0, k1      lw      k0, (k0)  /*处理函数*
/      nop      jr      k0        /*运行异常处理函数*/      nop        那
个异常处理程序表是如何初始化的呢?       在traps_init中,大家会看到
set_exception_vector(i,handler)这样的代码,      填的就是这张表啦.可是,如果你
用souce insigh之类的东西去找那个handler,往往      就落空了,??怎么没有
handle_ri,handle_tlbl..._?不着急,只不过是一个小trick,      还记得x86中断处
理的handler代码吗? 它们是用宏生成的:      entry.S      ...     #define
BUILD_HANDLER(exception,handler,clear,verbose)      .align  5;
NESTED(handle_##exception, PT_SIZE, sp);                    .set
noat;                                                SAVE_ALL;  /* 保
存现场,切换栈(如必要)*/
__BUILD_clear_##clear(exception); /*关中断?*/               .set
at;
__BUILD_##verbose(exception);                                jal
do_##handler;     /*干活*/                          move   a0, sp;
j       ret_from_exception;   /*回去*/                     nop;
END(handle_##exception)        /*生成处理函数*/
BUILD_HANDLER(adel,ade,ade,silent)              /* #4  */
BUILD_HANDLER(ades,ade,ade,silent)              /* #5  */
BUILD_HANDLER(ibe,ibe,cli,verbose)              /* #6  */
BUILD_HANDLER(dbe,dbe,cli,silent)               /* #7  */
BUILD_HANDLER(bp,bp,sti,silent)                 /* #9  */       认真追
```

究下去,这里的一些宏是很重要的,象SAVE_ALL(include/asm/stackframe.h),异常处理要高效,正确,这里要非常小心.这是因为硬件做的事情实在太少了. 别的暂时先不说了,下面我们来看外设中断(它是一种特殊的异常). entry.S并没有用BUILD_HANDLER生成中断处理函数,因为它是平台相关的 就以我的板子为例,在arch/mips/algor/p6032/kernel/中(标准内核没有) 增加了p6032IRQ.S这个汇编文件,里面定义了一个p6032IRQ的函数,它负责 鉴别中断源,调用相应的中断控制器处理代码,而在同目录的irq.c->init_IRQ
中调用set_except_vector(0,p6032IRQ)填表(所有的中断都引发异常0,并在cause寄存器中设置具体中断原因).

下面列出这两个文件以便解说:
p6032IRQ.s

algor p6032(我用的开发板)中断安排如下:
MIPS IRQ Source

```
*      --------        ------
*            0         Software (ignored)
*            1         Software (ignored)
*            2         bonito interrupt (hw0)
*            3         i8259A interrupt (hw1)
*            4         Hardware (ignored)
*            5         Debug Switch
*            6         Hardware (ignored)
*            7         R4k timer (what we use)

 .text
 .set    noreorder
 .set    noat
 .align  5
  NESTED(p6032IRQ, PT_SIZE, sp)

    SAVE_ALL /* 保存现场,切换堆栈(if usermode -> kernel mode)*/
    CLI      /* 关中断,mips有多种方法禁止响应中断,CLI用清status相应位
             的方法,如下:
              Move to kernel mode and disable interrupts.
              Set cp0 enable bit as sign that we're running
              on the kernel stack */
#define CLI
    mfc0    t0,CP0_STATUS;
    li      t1,ST0_CU0|0x1f;
    or      t0,t1;
    xori    t0,0x1f;
    mtc0    t0,CP0_STATUS
```

```
.set    at

mfc0    s0, CP0_CAUSE
/* get irq mask,中断响应时cause寄存器指示
   哪类中断发生
   注意,MIPS CPU只区分8个中断源,并没有象
   PC中从总线读取中断向量号,所以每类中断
   的代码要自己设法定位中断
 */

/* 挨个检查可能的中断原因 */
/* First we check for r4k counter/timer IRQ. */
andi    a0, s0, CAUSEF_IP7
beq     a0, zero, 1f
andi   a0, s0, CAUSEF_IP3      # delay slot, check 8259 interrupt
/* Wheee, a timer interrupt. */
li      a0, 63
jal     do_IRQ
move    a1, sp
j       ret_from_irq
nop                            # delay slot

1:      beqz    a0,1f
andi   a0, s0, CAUSEF_IP2

/* Wheee, i8259A interrupt. */
/* p6032也使用8259来处理一些pc style的设备*/
jal     i8259A_irqdispatch      /* 调用8259控制器的中断分派代码*/
move   a0, sp                  # delay slot

j       ret_from_irq
nop                            # delay slot

1:      beq     a0, zero, 1f
andi   a0, s0, CAUSEF_IP5

/* Wheee, bonito interrupt. */
/* bonito是6032板的北桥,它提供了一个中断控制器*/
jal     bonito_irqdispatch
move   a0, sp                  # delay slot
j       ret_from_irq
nop                            # delay slot

1:      beqz    a0,1f
```

```
    nop

    /* Wheee, a debug interrupt. */
    jal     p6032_debug_interrupt
    move   a0, sp                  # delay slot

    j       ret_from_irq
    nop                            # delay slot


    1:
    /* Here by mistake?  This is possible, what can happen
     * is that by the time we take the exception the IRQ
     * pin goes low, so just leave if this is the case.
     */
    j       ret_from_irq
    nop
END(p6032IRQ)
```

irq.c 部分代码如下:

p6032 中断共有四类:

begin{enumerate}

item timer 中断,单独处理

item debug 中断,单独处理

item 8259 中断,由 8259 控制器代码处理

item bonito 中断由 bonito 控制器代码处理

end{enumerate}

```
/* now mips kernel is using the same abstraction as x86 kernel,
   that is, all irq in the system are described in an struct
   array: irq_desc[]. Each item of a specific item records
   all the information about this irq,including status,action,
   and the controller that handle it etc. Below is the controller
   structure for bonito irqs,we can easily guess its functionality
   from its names.*/

hw_irq_controller bonito_irq_controller = {
    "bonito_irq",
    bonito_irq_startup,
    bonito_irq_shutdown,
    bonito_irq_enable,
    bonito_irq_disable,
    bonito_irq_ack,
```

```
    bonito_irq_end,
    NULL                    /* no affinity stuff for UP */
};

void
bonito_irq_init(u32 irq_base)
{
    extern irq_desc_t irq_desc[];
    u32 i;

    for (i= irq_base; i< P6032INT_END; i++) {
        irq_desc[i].status = IRQ_DISABLED;
        irq_desc[i].action = NULL;
        irq_desc[i].depth = 1;
        irq_desc[i].handler = &bonito_irq_controller;
    }

    bonito_irq_base = irq_base;
}

/* 中断初始化,核心的数据结构就是irq_desc[]数组
   它的每个元素对应一个中断,记录该中断的控制器类型,处理函数,状态等
   关于这些可以参见对x86中断的分析*/

void __init init_IRQ(void)
{
    Bonito;

    /*
     * Mask out all interrupt by writing "1" to all bit position in
     * the interrupt reset reg.
     */
    BONITO_INTEDGE = BONITO_ICU_SYSTEMERR | BONITO_ICU_MASTERERR
        | BONITO_ICU_RETRYERR | BONITO_ICU_MBOXES;
    BONITO_INTPOL = (1 << (P6032INT_UART1-16))
        | (1 << (P6032INT_ISANMI-16))
        | (1 << (P6032INT_ISAIRQ-16))
        | (1 << (P6032INT_UART0-16));

    BONITO_INTSTEER = 0;
    BONITO_INTENCLR = ~0;

    /* init all controllers */
    init_generic_irq();
```

```
    init_i8259_irqs();
    bonito_irq_init(16);

    BONITO_INTSTEER |= 1 << (P6032INT_ISAIRQ-16);
    BONITO_INTENSET = 1 <<  (P6032INT_ISAIRQ-16);

    /* hook up the first-level interrupt handler */
    set_except_vector(0, p6032IRQ);

    ...
}

/*p6032IRQ 发现一个 bonito 中断后调用这个*/
asmlinkage void
bonito_irqdispatch(struct pt_regs *regs)
{
    Bonito;

    int irq;
    unsigned long int_status;
    int i;

    /* Get pending sources, masked by current enables */
        /* 到底是哪个中断呢?从主板寄存器读*/
        int_status = BONITO_INTISR & BONITO_INTEN & ~(1 <<
(P6032INT_ISAIRQ-16))
                ;

        /* Scan all pending interrupt bits and execute appropriate
actions */
        for (i=0; i<32 && int_status; i++) {
                if (int_status & 1<<i) {
                        irq = i + 16; /* 0-15 assigned to 8259int,16-
48 bonito*/
                                /* Clear bit to optimise loop exit */
                                int_status &= ~(1<<i);
                        do_IRQ(irq,regs);

                }
        }

        return;
}
```

8259 控制器的代码类似,不再列出.

更高层一点的通用 irq 代码在 arch/mips/kernel/irq.c
arch/mips/kernel/i8259.c

总之,p6032 上一个中断的过程是:

1.外设发出中断,通过北桥在 cpu 中断引脚上(mips CPU 有多个中断引脚)引起异常
2. cpu 自动跳转到 0x80000180 的通用异常入口,根据 cause 寄存器查表找到中断处理函数入口 p6032IRQ
3. p6032IRQ 保存上下文,识别中断类别,把中断转交给相应的中断控制器
4. 中断控制器的代码进一步识别出具体的中断号,做出相应的应答并调用中断处理 do_irq

现在还有不少平台没有使用这种 irq_desc[],controller,action 的代码,阅读的时候
可能要注意.

上次说道 SAVE_ALL 里有些玄机,这里把 include/asm-mips/stackframe.h 对着注解一下,希望能说清楚一些.
(因为时间关系,我写的文档将主要以这种文件注解为主,加上我认为有用的背景知识或者分析.)

```
/*
 一些背景知识

 一.mips 汇编有个约定(后来也有些变化,我们不管,o32,n32),32 个通用寄存器不是一视同仁
  的,而是分成下列部分:
        寄存器号              符号名              用途
          0                   始终为 0      看起来象浪费,其实很有用
          1                   at            保留给汇编器使用
          2-3                 v0,v1         函数返回值
          4-7                 a0-a3         前头几个函数参数
          8-15                t0-t7         临时寄存器,子过程可以不保存就使用
          24-25               t8,t9         同上
          16-23               s0-s7         寄存器变量,子过程要使用它必须先保存
                                            然后在退出前恢复以保留调用者需要的值
          26,27               k0,k1         保留给异常处理函数使用
          28                  gp            global pointer;用于方便存取全局或
者静态变量
          29                  sp            stack pointer
```

30 s8/fp 第9个寄存器变量;子过程可以用它做frame pointer
31 ra 返回地址

硬件上这些寄存器并没有区别(除了0号),区分的目的是为了不同的编译器产生的代码可以通用

二. r4k MIPS CPU中和异常相关的控制寄存器(这些寄存器由协处理器cp0控制,有独立的存取方法)有:

1.status 状态寄存器

```
31  28 27 26 25 24        16 15          8 7 6  5 4 3  2   1   0
-----------------------------------------------------------------
| cu0-3|RP|FR|RE| Diag Status|   IM7-IM0  |KX|SX|UX|KSU|ERL|EXL|IE|
-----------------------------------------------------------------
```

其中KSU,ERL,EXL,IE位在这里很重要:

KSU: 模式位 00 -kernel 01--Supervisor 10--User

ERL: error level,0->normal,1->error

EXL: exception level,0->normal,1->exception,异常发生是EXL自动置1

IE: interrupt Enable, 0 -> disable interrupt,1->enable interrupt

(IM位则可以用于enbale/disable具体某个中断,ERL||EXL=1 也使得中断不能响应)

系统所处的模式由KSU,ERL,EXL决定:

User mode: KSU = 10 && EXL=0 && ERL=0

Supervisor mode(never used): KSU=01 && EXL=0 && ERL=0

Kernel mode: KSU=00 || EXL=1 || ERL=1

2.cause寄存器

```
31 30 29 28 27          16 15           8 7 6          2  1  0
-----------------------------------------------------------------
|BD|0 | CE  |     0        | IP7 - IP0    |0|Exc code     | 0 |
-----------------------------------------------------------------
```

异常发生时cause被自动设置

其中:

BD指示最近发生的异常指令是否在delay slot中

CE发生coprocessor unusable异常时的coprocessor编号(mips有4个cp)

```
        IP: interrupt pending, 1->pending,0->no interrupt,CPU有6个中
断
            引脚,加上两个软件中断(最高两个)
        Exc code:异常类型,所有的外设中断为0,系统调用为8,...
      3.EPC
        对一般的异常,EPC包含:
          . 导致异常的指令地址(virtual)
          or. if 异常在delay slot指令发生,该指令前面那个跳转指令的地址
        当EXL=1时,处理器不写EPC
      4.和存储相关的:
        context,BadVaddr,Xcontext,ECC,CacheErr,ErrorEPC
        以后再说

      一般异常处理程序都是先保存一些寄存器,然后清除EXL以便嵌套异常,
      清除KSU保持核心态,IE位看情况而定;处理完后恢复一些保存内容以及CPU状态

  */

/* SAVE_ALL 保存所有的寄存器,分成几个部分,方便不同的需求选用*/
/*保存AT寄存器,sp是栈顶PT_R1是at寄存器在pt_regs结构的偏移量
  .set xxx是汇编指示,告诉汇编器要干什么,不要干什么,或改变状态
 */
#define SAVE_AT                                         \
                .set    push;                           \
                .set    noat;                           \
                sw      $1, PT_R1(sp);                  \
                .set    pop

/*保存临时寄存器,以及hi,lo寄存器(用于乘法部件保存64位结果)
 可以看到mfhi(取hi寄存器的值)后并没有立即保存,这是因为
 流水线中,mfhi的结果一般一拍不能出来,如果下一条指令就想
 用v1则会导致硬件停一拍,这种情况下让无关的指令先做可以提高
 效率.下面还有许多类似的例子
 */
#define SAVE_TEMP                                       \
                mfhi    v1;                             \
                sw      $8, PT_R8(sp);                  \
                sw      $9, PT_R9(sp);                  \
                sw      v1, PT_HI(sp);                  \
                mflo    v1;                             \
                sw      $10,PT_R10(sp);                 \
                sw      $11, PT_R11(sp);                \
                sw      v1,  PT_LO(sp);                 \
                sw      $12, PT_R12(sp);                \
```

```
                sw      $13, PT_R13(sp);                \
                sw      $14, PT_R14(sp);                \
                sw      $15, PT_R15(sp);                \
                sw      $24, PT_R24(sp)

/* s0-s8 */
#define SAVE_STATIC                                      \
                sw      $16, PT_R16(sp);                \
                sw      $17, PT_R17(sp);                \
                sw      $18, PT_R18(sp);                \
                sw      $19, PT_R19(sp);                \
                sw      $20, PT_R20(sp);                \
                sw      $21, PT_R21(sp);                \
                sw      $22, PT_R22(sp);                \
                sw      $23, PT_R23(sp);                \
                sw      $30, PT_R30(sp)

#define __str2(x) #x
#define __str(x) __str2(x)



/*ok,下面对这个宏有冗长的注解*/
#define save_static_function(symbol)
\
__asm__ (
\
        ".globl\t" #symbol "\n\t"
\
        ".align\t2\n\t"
\
        ".type\t" #symbol ", @function\n\t"
\
        ".ent\t" #symbol ", 0\n"
\
        #symbol":\n\t"
\
        ".frame\t$29, 0, $31\n\t"
\
        "sw\t$16,"__str(PT_R16)"($29)\t\t\t#
save_static_function\n\t"  \
        "sw\t$17,"__str(PT_R17)"($29)\n\t"
\
```

```
        "sw\t$18,"__str(PT_R18)"($29)\n\t"
\
        "sw\t$19,"__str(PT_R19)"($29)\n\t"
\
        "sw\t$20,"__str(PT_R20)"($29)\n\t"
\
        "sw\t$21,"__str(PT_R21)"($29)\n\t"
\
        "sw\t$22,"__str(PT_R22)"($29)\n\t"
\
        "sw\t$23,"__str(PT_R23)"($29)\n\t"
\
        "sw\t$30,"__str(PT_R30)"($29)\n\t"
\
        ".end\t" #symbol "\n\t"
\
        ".size\t" #symbol",. - " #symbol)

/* Used in declaration of save_static functions.  */
#define static_unused static __attribute__((unused))

/*以下这一段涉及比较微妙的问题,没有兴趣可以跳过*/

/* save_static_function 宏是一个令人迷惑的东西,它定义了一个汇编函数,保存 s0-s8
   可是这个函数没有返回!实际上,它只是一个函数的一部分:
   在 arch/mips/kernel/signal.c 中有:
       save_static_function(sys_rt_sigsuspend);
       static_unused int
       _sys_rt_sigsuspend(struct pt_regs regs)
       {
               sigset_t *unewset, saveset, newset;
                       size_t sigsetsize;
   这里用 save_static_function 定义了 sys_rt_sigsuspend,而实际上如果
   你调用 sys_rt_sigsuspend 的话,它保存完 s0-s8 后,接着就调用
_sys_rt_sigsuspend!
   看它链接后的反汇编片段:
      80108cc8 <sys_rt_sigsuspend>:
      80108cc8:       afb00058        sw      $s0,88($sp)
      80108ccc:       afb1005c        sw      $s1,92($sp)
      80108cd0:       afb20060        sw      $s2,96($sp)
      80108cd4:       afb30064        sw      $s3,100($sp)
      80108cd8:       afb40068        sw      $s4,104($sp)
      80108cdc:       afb5006c        sw      $s5,108($sp)
      80108ce0:       afb60070        sw      $s6,112($sp)
```

```
80108ce4:       afb70074        sw      $s7,116($sp)
80108ce8:       afbe0090        sw      $s8,144($sp)

80108cec <_sys_rt_sigsuspend>:
80108cec:       27bdffc8        addiu   $sp,$sp,-56
80108cf0:       8fa80064        lw      $t0,100($sp)
80108cf4:       24030010        li      $v1,16
80108cf8:       afbf0034        sw      $ra,52($sp)
80108cfc:       afb00030        sw      $s0,48($sp) ---> notice
80108d00:       afa40038        sw      $a0,56($sp)
80108d04:       afa5003c        sw      $a1,60($sp)
80108d08:       afa60040        sw      $a2,64($sp)
...
```

用到 save_static_function 的地方共有 4 处:

```
signal.c:save_static_function(sys_sigsuspend);
signal.c:save_static_function(sys_rt_sigsuspend);
syscall.c:save_static_function(sys_fork);
syscall.c:save_static_function(sys_clone);
```

我们知道 s0-s8 如果在子过程用到,编译器本来就会保存/恢复它的(如上面的 s0),
那为何要搞这个花招呢?我分析之后得出如下结论:
(警告:以下某些内容是我的推测,可能不完全正确)

先看看 syscall 的处理,syscall 也是 mips 的一种异常,异常号为 8.上次我们说了一般异常是如何工作的,但在 handle_sys 并非用 BUILD_HANDLER 生成,而是在 scall_o23.S 中定义,因为它又有其特殊之处.

1.缺省情况它只用了 SAVE_SOME,并没有保存 at,t*,s*等寄存器,因为 syscall
是由应用程序调用的,不象中断,任何时候都可以发生,所以一般编译器就可以保证不会丢数据了(at,t*的值应该已经无效,s*的值会被函数保存恢复).
这样可以提高系统调用的效率

2.它还得和用户空间打交道(取参数,送数据)

还有个别系统调用需要在特定的时候手工保存 s*寄存器,如上面的几个.为什么呢?
对 sigsuspend 来说,它将使进程在内核中睡眠等待信号到来,信号来了之后将直接先回到进程的信号处理代码,而信号处理代码可能希望看到当前进程的寄存器(sigcontext),这是通过内核栈中的 pt_regs 结构获得的,所以内核必需把 s*寄存器保存到 pt_regs 中.对于 fork 的情况,则似乎是为了满足 vfork 的要求.(vfork 时,子进程
不拷贝页表(即和父进程完全共享内存),注意,连 copy-on-write 都没有!父进程挂起一直到子进程不再使用它的资源(exec 或者 exit)).fork 系统调用使用 ret_from_fork 返回,其中调用到了 RESTORE_ALL_AND_RET(entry.S),需要恢复 s*.

这里还有一个很容易混乱的地方: 在scall_o32.S和entry.S中有几个函数(汇编)是同名
的,如restore_all,sig_return等.总体来说scall_o32.S中是对满足o32(old 32bit)汇编
约定的系统调用处理,可以避免保存s*,而entry.S中是通用的,保存/恢复所由寄存器
scall_o32.S中也有一些情况需要保存静态寄存器s*,此时它就会到ret_from_syscall
而不是本文件中的o32_ret_from_syscall返回了,两者的差别就是恢复的寄存器数目
不同.scall_o32.S中一些错误处理直接用ret_from_syscall返回,我怀疑会导致s*寄存器
被破坏,有机会请各路高手指教.

好了,说了一通系统调用,无非是想让大家明白内核中寄存器的保存恢复过程,以及
为了少做些无用功所做的努力.下面看为什么要save_static_function:为了避免
s0寄存器的破坏.
如果我们使用

```
sys_rt_sigsuspend()
{   ..
   save_static;
   ...
}
```

会有什么问题呢,请看,

```
Nasty degree - 3 days of tracking.

The symptom was pthread cannot be created.  In the end the caller will
get a BUS error.

What exactly happened has to do with how registers are saved. Below
attached is the beginning part of sys_sigsuspend() function.  It is easy
to see that s0 is saved into stack frame AFTER its modified.  Next time
when process returns to userland, the s0 reg will be wrong!

So the bug is either

1) that we need to save s0 register in SAVE_SOME and not save it in
save_static; or that
```

```
    2) we fix compiler so that it does not use s0 register in that
case (it
            does the same thing for sys_rt_sigsuspend)

    I am sure Ralf will have something to say about it.  :-)  In any
case, I
    attached a patch for 1) fix.

sys_sigsuspend(struct pt_regs regs)
{
        8008e280:   27bdffc0        addiu   $sp,$sp,-64
        8008e284:   afb00030        sw      $s0,48($sp)
                   sigset_t *uset, saveset, newset;

                   save_static(&regs);
        8008e288:   27b00040        addiu   $s0,$sp,64   /*
save_static 时
                                                          s0 已经破坏
*/
        8008e28c:   afbf003c        sw      $ra,60($sp)
        8008e290:   afb20038        sw      $s2,56($sp)
        8008e294:   afb10034        sw      $s1,52($sp)
        8008e298:   afa40040        sw      $a0,64($sp)
        8008e29c:   afa50044        sw      $a1,68($sp)
        8008e2a0:   afa60048        sw      $a2,72($sp)
        8008e2a4:   afa7004c        sw      $a3,76($sp)
        8008e2a8:   ae100058        sw      $s0,88($s0)
        8008e2ac:   ae11005c        sw      $s1,92($s0)

#ifdef CONFIG_SMP
#  define GET_SAVED_SP                                  \
                mfc0    k0, CP0_CONTEXT;                \
                lui     k1, %hi(kernelsp);              \
                srl     k0, k0, 23;                     \
               sll     k0, k0, 2;                      \
                addu    k1, k0;                         \
                lw      k1, %lo(kernelsp)(k1);

#else
#  define GET_SAVED_SP                                  \
/*实际上就是 k1 = kernelsp, kernelsp 保存当前进程的内核栈指针 */
               lui     k1, %hi(kernelsp);              \
               lw      k1, %lo(kernelsp)(k1);
#endif
```

```
/*判断当前运行态,设置栈顶sp
  保存寄存器--参数a0-a3:4-7,返回值v0-v1:2-3,25,28,31以及一些控制寄存器,
  */
#define SAVE_SOME                                       \
                .set    push;                           \
                .set    reorder;                        \
                mfc0    k0, CP0_STATUS;                 \
                sll     k0, 3;     /* extract cu0 bit */ \
                .set    noreorder;                      \
                bltz    k0, 8f;                         \
                 move   k1, sp;                         \
                .set    reorder;                        \
                /* Called from user mode, new stack. */  \
                 GET_SAVED_SP                            \
8:                                                      \
                move    k0, sp;                         \
                subu    sp, k1, PT_SIZE;                \
                sw      k0, PT_R29(sp);                 \
                 sw     $3, PT_R3(sp);                  \
                sw      $0, PT_R0(sp);                 \
                mfc0    v1, CP0_STATUS;                 \
                sw      $2, PT_R2(sp);                  \
                sw      v1, PT_STATUS(sp);              \
                sw      $4, PT_R4(sp);                  \
                mfc0    v1, CP0_CAUSE;                  \
                sw      $5, PT_R5(sp);                  \
                sw      v1, PT_CAUSE(sp);               \
                sw      $6, PT_R6(sp);                  \
                mfc0    v1, CP0_EPC;                    \
                sw      $7, PT_R7(sp);                  \
                sw      v1, PT_EPC(sp);                 \
                sw      $25, PT_R25(sp);                \
                sw      $28, PT_R28(sp);                \
                sw      $31, PT_R31(sp);                \
                ori     $28, sp, 0x1fff;                \
                xori    $28, 0x1fff;                    \
                .set    pop

#define SAVE_ALL                                        \
                SAVE_SOME;                              \
                SAVE_AT;                                \
                SAVE_TEMP;                              \
                SAVE_STATIC
```

```
#define RESTORE_AT                                      \
                .set    push;                           \
                .set    noat;                           \
                lw      $1,  PT_R1(sp);                 \
                .set    pop;

#define RESTORE_TEMP                                    \
                lw      $24, PT_LO(sp);                 \
                lw      $8, PT_R8(sp);                  \
                lw      $9, PT_R9(sp);                  \
                mtlo    $24;                            \
                lw      $24, PT_HI(sp);                 \
                lw      $10,PT_R10(sp);                 \
                lw      $11, PT_R11(sp);                \
                mthi    $24;                            \
                lw      $12, PT_R12(sp);                \
                lw      $13, PT_R13(sp);                \
                lw      $14, PT_R14(sp);                \
                lw      $15, PT_R15(sp);                \
                lw      $24, PT_R24(sp)

#define RESTORE_STATIC                                  \
                lw      $16, PT_R16(sp);                \
                lw      $17, PT_R17(sp);                \
                lw      $18, PT_R18(sp);                \
                lw      $19, PT_R19(sp);                \
                lw      $20, PT_R20(sp);                \
                lw      $21, PT_R21(sp);                \
                lw      $22, PT_R22(sp);                \
                lw      $23, PT_R23(sp);                \
                lw      $30, PT_R30(sp)

#if defined(CONFIG_CPU_R3000) || defined(CONFIG_CPU_TX39XX)

#define RESTORE_SOME                                    \
                .set    push;                           \
                .set    reorder;                        \
                mfc0    t0, CP0_STATUS;                 \
                .set    pop;                            \
                ori     t0, 0x1f;                       \
                xori    t0, 0x1f;                       \
                mtc0    t0, CP0_STATUS;                 \
                li      v1, 0xff00;                     \
```

```
                and     t0, v1;                         \
                lw      v0, PT_STATUS(sp);               \
                nor     v1, $0, v1;                     \
                and     v0, v1;                         \
                or      v0, t0;                         \
                mtc0    v0, CP0_STATUS;                  \
                lw      $31, PT_R31(sp);                 \
                lw      $28, PT_R28(sp);                 \
                lw      $25, PT_R25(sp);                 \
                lw      $7,  PT_R7(sp);                  \
                lw      $6,  PT_R6(sp);                  \
                lw      $5,  PT_R5(sp);                  \
                lw      $4,  PT_R4(sp);                  \
                lw      $3,  PT_R3(sp);                  \
                lw      $2,  PT_R2(sp)

#define RESTORE_SP_AND_RET                               \
                .set    push;                           \
                .set    noreorder;                      \
                lw      k0, PT_EPC(sp);                  \
                lw      sp,  PT_R29(sp);                 \
                jr      k0;                              \
                 rfe;                                   \
                 ^^^^^
/* 异常返回时,把控制转移到用户代码和把模式从内核态改为用户态要同时完成
   如果前者先完成,用户态指令有机会以内核态运行导致安全漏洞;
   反之则会由于用户态下不能修改状态而导致异常
   r3000 以前使用 rfe(restore from exception)指令,这个指令把 status 寄存器
   状态位修改回异常发生前的状态(利用硬件的一个小堆栈),但不做跳转.我们使用一个
   技巧来完成要求:在一个跳转指令的 delay slot 中放 rte.因为 delay slot 的指令
   是一定会做的,跳转完成时,status 也恢复了.
   MIPS III(r4000)以上的指令集则增加了 eret 指令来完成整个工作: 它清除
   status 寄存器的 EXL 位并跳转到 epc 指定的位置.
*/

                .set    pop

#else

#define RESTORE_SOME                                     \
                .set    push;                           \
                .set    reorder;                        \
                mfc0    t0, CP0_STATUS;                 \
                .set    pop;                            \
```

```
                ori     t0, 0x1f;                       \
                xori    t0, 0x1f;                       \
                mtc0    t0, CP0_STATUS;                 \
                li      v1, 0xff00;                     \
                and     t0, v1;                        \
                lw      v0, PT_STATUS(sp);              \
                nor     v1, $0, v1;                    \
                and     v0, v1;                        \
                or      v0, t0;                        \
                mtc0    v0, CP0_STATUS;                 \
                lw      v1, PT_EPC(sp);                 \
                mtc0    v1, CP0_EPC;                    \
                lw      $31, PT_R31(sp);                \
                lw      $28, PT_R28(sp);                \
                lw      $25, PT_R25(sp);                \
                lw      $7,  PT_R7(sp);                 \
                lw      $6,  PT_R6(sp);                 \
                lw      $5,  PT_R5(sp);                 \
                lw      $4,  PT_R4(sp);                 \
                lw      $3,  PT_R3(sp);                 \
                lw      $2,  PT_R2(sp)

#define RESTORE_SP_AND_RET                               \
                lw      sp,  PT_R29(sp);                \
                .set    mips3;                         \
                eret;                                  \
                .set    mips0

#endif

#define RESTORE_SP                                       \
                lw      sp,  PT_R29(sp);                \

#define RESTORE_ALL                                      \
                RESTORE_SOME;                           \
                RESTORE_AT;                             \
                RESTORE_TEMP;                           \
                RESTORE_STATIC;                         \
                RESTORE_SP

#define RESTORE_ALL_AND_RET                              \
                RESTORE_SOME;                           \
                RESTORE_AT;                             \
                RESTORE_TEMP;                           \
```

```
                RESTORE_STATIC;                                 \
                RESTORE_SP_AND_RET


/*
 * Move to kernel mode and disable interrupts.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define CLI                                                     \
                mfc0    t0,CP0_STATUS;                  \
                li      t1,ST0_CU0|0x1f;                \
                or      t0,t1;                          \
                xori    t0,0x1f;                        \
                mtc0    t0,CP0_STATUS

/*
 * Move to kernel mode and enable interrupts.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define STI                                                     \
                mfc0    t0,CP0_STATUS;                  \
                li      t1,ST0_CU0|0x1f;                \
                or      t0,t1;                          \
                xori    t0,0x1e;                        \
                mtc0    t0,CP0_STATUS

/*
 * Just move to kernel mode and leave interrupts as they are.
 * Set cp0 enable bit as sign that we're running on the kernel stack
 */
#define KMODE                                                   \
                mfc0    t0,CP0_STATUS;                  \
                li      t1,ST0_CU0|0x1e;                \
                or      t0,t1;                          \
                xori    t0,0x1e;                        \
                mtc0    t0,CP0_STATUS

#endif /* __ASM_STACKFRAME_H */
```

下面是我在为 godson CPU 的页面可执行保护功能增加内核支持时分析 linux-mips mmu 实现的一些笔记,没有时间整理,有兴趣就看看吧.也许第 5 节对整个工作过程的分析会有些用,其它语焉不详的东西多数只是对我本人有点用.

首先的,关键的,要明白 MIPS CPU 的 tlb 是软件管理的,cache 也不是透明的,具体的

参见它们的用户手册.

(for sgi-cvs kernel 2.4.17)

1. mmu context

cpu 用 8 位 asid 来区分 tlb 表项所属的进程,但是进程超过 256 个怎么办?

linux 实现的思想是软件扩展,每 256 个一组,TLB 任何时候只存放同一组的 asid

因此不会冲突. 从一组的某个进程切换到另一组时,把 tlb 刷新

ASID switch

include/asm/mmu_context.h:

asid_cache:

8bit physical asid + software extension, the software extension bits

are used as a version; this records the newest asid allocated,while

process->mm->context records its own version.

get_new_mmu_context:

asid_cache++, if increasement lead to change of software extension part

then flush icache & tlb to avoid conflicting with old versions.

asid_cache = 0 reserved to represent no valid mmu context case,so the

first asid_cache version start from 0x100.

switch_mm:

if asid version of new process differs from current process',get a new

context for it.(it's safe even if it gets same 8bit asid as previous

because this process' tlb entries must have been flushed at the time

of version increasement)

set entryhi,install pgd

activate_mm:

get new asid,set to entryhi,install pgd.

2. pte bits

页表的内容和 TLB 表项关系

entrylo[01]:

```
3130 29                      6  5  3 2 1 0
-----------------------------------------
|   |     PFN                  | C  |D|V|G|
-----------------------------------------
```

r4k pte:

```
31               12 111098 7 6 5  3 2 1 0
-----------------------------------------
```

```
        |    PFN            | C  |D|V|G|B|M|A|W|R|P|
        -----------------------------------------
    C: cache attr.
    D: Dirty
    V: valid
    G: global
    B: R4K_BUG
    M: Modified
    A: Accessed
    W: Write
    R: Read
    P: Present
    (last six bits implemented in software)

   godson entrylo:
       bit 30 is used as execution protect bit E,only bit25-6 are
used
       as PFN.
       instruction fetch from a page has E cleared lead to address
error
       exception.
   godson pte:
       31                12 111098 7 6 5  3 2 1 0
       -----------------------------------------
       |    PFN            | C  |D|V|G|E|M|A|W|R|P|
       -----------------------------------------
   E: software implementation of execute protection.Page is
executable when
      E is set,non-executable otherwise.(Notice,it is different from
hardware
      bit 30 in entrylo)

3. actions dealing with pte
  pte_page: get page struct from its pte value
  pte_{none,present,read,write,dirty,young}: get pte status,use
software bits
  pte_wrprotect: &= ~(_PAGE_WRITE | _PAGE_SILENT_WRITE)
  pte_rdprotect: &= ~(_PAGE_READ | _PAGE_SILENT_READ)
  pte_mkclean: &= ~(_PAGE_MODIFIED | _PAGE_SILENT_WRITE)
  pte_mkold: &= ~(_PAGE_ACCESSED | _PAGE_SILENT_READ)
  pte_mkwrite: |= _PAGE_WRITE && if (_PAGE_MODIFIED) |=
_PAGE_SILENT_WRITE
  pte_mkread: |= _PAGE_READ && if (_PAGE_ACCESSED) |=
_PAGE_SILENT_READ
```

```
  pte_mkdirty: |= _PAGE_MODIFIED && if (_PAGE_WRITE) |=
_PAGE_SILENT_WRITE
  pte_mkyoung: |= _PAGE_ACCESSED && if (_PAGE_READ) |=
_PAGE_SILENT_READ
  pgprot_noncached: (&~CACHE_MASK) | (_CACHE_UNCACHED)
  mk_pte(page,prot): (unsigned long) ( page - memmap ) << PAGE_SHIFT
| prot
  mk_pte_phys(physpage,prot): physpage | prot
  pte_modify(pte,prot): ( pte & _PAGE_CHG_MASK ) | newprot
  page_pte_{prot}: unused?
  set_pte: *ptep = pteval
  pte_clear: set_pte(ptep,__pte(0));

  ptep_get_and_clear

  pte_alloc/free


4. exceptions
  tlb refill exception(0x80000000):
     (1) get badvaddr,pgd
     (2) pte table ptr = badvaddr>>22 < 2 + pgd ,
     (3) get context,offset = context >> 1 & 0xff8 (bit 21-13 + three
zero),
     (4) load offset(pte table ptr) and offset+4(pte table ptr),
     *(5) right shift 6 bits,write to entrylo[01],
     (6) tlbwr
  tlb modified exception(handle_mod):
     (1) load pte,
     *(2) if _PAGE_WRITE set,set ACCESSED | MODIFIED | VALID | DIRTY,
          reload tlb,tlbwi
        else DO_FAULT(1)
  tlb load exception(handle_tlbl):
     (1) load pte
     (2) if _PAGE_PRESENT && _PAGE_READ, set ACCESSED | VALID
         else DO_FAULT(0)
  tlb store exception(handle_tlbs):
     (1) load pte
     *(2) if _PAGE_PRESENT && _PAGE_WRITE,set ACCESSED | MODIFIED |
VALID | DIRTY
         else DO_FAULT(1)

   items marked with * need modification.
```

```
5. protection_map
   all _PXXX map to page_copy? Although vm_flags will at last make
pte writeable
   as needed,but will this be inefficient? it seems that alpha is not
doing so.

   mm setup/tear down:
     on fork,copy_mm:
       allocate_mm,
       memcpy(new,old)
                          slow path
       mm_init-->pgd_alloc-->pgd_init-->all point to invalid_pte
                            -->copy kseg pgds from init_mm
                         fast path: what's the content of pgd?
                           --> point to invalid_pte too,see
clear_page_tables
       dup_mmap->copy_page_range-->alloc page table entries and do
cow if needed.
       copy_segmens--null
       init_new_context--set mm->context=0(allocate an array for SMP
first)
     on exec(elf file),load_elf_binary:
       flush_old_exec:
         exec_mmap
            exit_mmap(old_mm)
               free vm_area_struct
               zap_page_range: free pages
               clear_page_tables
                 pgd_clear: do nothing
                 pmd_clear: set to invalid_pte
                 pte_clear: set to zero
            mm_alloc
            initialize new mm( init_new_contex,add to list,activate
it)
            mmput(oldmm)
      setup_arg_pages:
        initialize stack segment. mm_area_struct for stack segment is
setup
        here.
      load elf image into the correct location in memory
         elf_prot generated from eppnt->p_flags
         elf_map(..,elf_prot,..)
           do_mmap
```

```
   a typical session for a user page to be read then written:
     (1) user allocates the space
     (2) kernel call do_mmap/do_brk, vm_area_struct created
     (3) user tries to read
     (4) tlb refill exception occurs,invalid_pte_table's entry is
loaded into
         tlb
     (5) tlbl exception occurs,
          do_page_fault(0)->handle_mm_fault(allocate pte_table)-
>handle_pte_fault
          -->do_no_page-->map to ZERO
page,readonly,set_pte,update_mmu_cache
          (update_mmu_cache put new pte to tlb,NEED change for
godson)
     (6) read done,user tries to write
     (7) tlbs exception occurs(suppose the tlb entry is not yet
kicked out)
         because pte is write protected,do_page_fault(1) called.
         handle_mm_fault(find out the pte)-->handle_pte_fault-
>do_wp_page
         -->allocate page,copy page,break_cow-->make a writeable pte,
         -->establish_pte-->write pte and update_mmu_cache
     (8) write done.

   above has shown that handle_mm_fault doesn't care much about what
the
   page_prot is. (Of course,it has to be reasonable)
   What really matters is vm_flags,it will decide whether an access
is valid

6. do_page_fault
   seems ok
7. swapping
   seems ok

8. adding execution protection
 2002.3.16:
      TLB execute protection bit support.
        1. generic support
        idea:
          use bit 5 in pte to maintain a software version of
_PAGE_EXEC
          modify TLB refill code to reflect it into hardware bit(bit
30)
```

```
        affected files:
          include/asm/pgtable.h:
            define _PAGE_EXEC
            change related PAGE_XXX macros and protection_map
            add pte_mkexec/pte_exprotect
            add godson_mkexec/godson_mkprotect
          arch/mips/mm/tlbex-r4k.S:
           tlb_refill exception & PTE_RELOAD macro:
            test bit 5 and translated it into bit30 in entrylo
            using godson's cp0 register 23/24 as temporary store
place
            Note: bit5 and bit30 have adverse meaning,bit5 set==bit30
                  cleared==page executable,
          arch/mips/mm/tlb-r4k.c:
            update_mmu_cache:
                test bit 5 and translated it into bit30 in entrylo
             implement godson_mkexec/godson_exprotect
          arch/mips/config.in:
            add option CONFIG_CPU_HAS_EXECUTE_PROTECTION
        2. non-executable stack support
          interface:
           by default no protection is taken,To take advantage of
           this support,one should call sysmips syscall to set the
           flag bit and then execute the target program.
          affected files:
           include/asm/processor.h:
                define MF_STACK_PROTECTION flag
           fs/exec.c:
                judge which protection to use
           arch/mips/kernel/signal.c:
                enable/disable execute for signal trampoline
           arch/mips/math-emu/cp1emu.c:
                enable/disable execute for delay slot emulation
trampoline
           arch/mips/kernel/sysmips.c:
                handle MF_STACK_PROTECTION
```