

<http://blog.sina.com.cn/u/2944286143> [订阅] [手机订阅]



[首页](#) | [博文目录](#) | [图片](#) | [关于我](#)

字体大小: 大 中 小

+ 转载 ▼

作为后期学习的一点总结性的东西

MIPS指令特点:

- 1、所有指令都是32位编码；
- 2、有些指令有26位供目标地址编码；有些则只有16位。因此要想加载任何一个32位值，就得用两个加载指令。16位的目标地址意味着，指令的跳转或子函数的位置必须在64K以内（上下32K）；
- 3、所有的动作原理上要求必须在1个时钟周期内完成，一个动作一个阶段；
- 4、有32个通用寄存器，每个寄存器32位（对32位机）或64位（对64位机）；
- 5、本身没有任何帮助运算判断的标志寄存器，要实现相应的功能时，是通过测试两个寄存器是否相等来完成的；
- 6、所有的运算都是基于32位的，没有对字节和对半字的运算（MIPS里，字定义为32位，半字定义为16位）；
- 7、没有单独的栈指令，所有对栈的操作都是统一的内存访问方式。因为push和pop指令实际上是一个复合操作，包含对内存的写入和对栈指针的移动；
- 8、由于MIPS固定指令长度，所以造成其编译后的二进制文件和内存占用空间比x86的要大，（x86平均指令长度只有3个字节多一点，而MIPS是4个字节）；
- 9、寻址方式：只有一种内存寻址方式。就是基地址加一个16位的地址偏移；
- 10、内存中的数据访问必须严格对齐（至少4字节对齐）

11、跳转指令只有26位目标地址，再加上2位的对齐位，可寻址28位的空间，即256M。意思即是说，在一个C程序内，goto语句只能跳转到它之前的128M和之后的128M这个地址空间之内

12、条件分支指令只有16位跳转地址，加上2位的对齐位，共18位寻址空间，即256K。意思即是说，在一个C程序内，if语句只能跳转到它之前的128K和之后的128K这个地址空间之内；

13、MIPS默认不把子函数的返回地址（就是调用函数的受害指令地址）存放到栈中，而是存放到\$31寄存器中；这对那些叶子函数有利。如果遇到嵌套的函数的话，有另外的机制处理：

14、流水线效应。由于采用了高度的流水线，结果产生了一些对程序员来说可见的效应，需要注意。最重要的两个效应就是分支延迟效应和载入延迟效应。

a 任何一个分支跳转语句后面的那条语句叫做分支延迟槽。实际上在程序执行到分支语句时，当他刚把要跳转到的地址填充好（到代码计数器里），还没完成本条指令，分支语句后面的那个指令就执行了。这是因为流水线效应，几条指令同时在执行，只是处于不同的阶段。具体看书上说提前半条指令执行，没看懂。分支延迟槽常用被利用起来完成一些参数初始化等相关工作，而不是被浪费了。

 b 载入延迟是这样的。当执行一条从内存中载入数据的命令时，是先载入到高速缓冲中，然后再取到寄存器中，这个过程相对来说是比较慢的。在这个过程完成之前，可能已经有几条在流水线上的指令被执行了。这几条在载入指令后被执行的指令就被称作载入延迟槽。现在就有一个问题，如果后面这几条指令要用到载入指令所载入的那个数据怎么办？一个通用的办法是，把内部锁加在数据载入过程上，这样，当后面的指令要用这条指令时，就只有先停止运行（在ALU阶段），等这条数据载入指令完成了后再开始运行。

* MIPS指令的五级流水线：每条指令都包含五个执行阶段。

第一阶段：从指令缓冲区中取指令。占一个时钟周期：

第二阶段：从指令中的源寄存器域（可能有两个）的值（为一个数字，指定\$0~\$31中的某一个）所代表的寄存器中读出数据。占半个时钟周期；

第三阶段：在一个时钟周期内做一次算术或逻辑运算。占一个时钟周期；

第四阶段：指令从数据缓冲中读取内存变量的阶段。从平均来讲，大约有3/4的指令在这个阶段没做什么事情，但它是指令有序性的保证（为什么是保证，我还没看清楚？）。占一个时钟周期：

 微博

发纸条

加关注

博客等级:

博客积分: 0

博客访问: **1,881**

关注人气: 0

获赠金笔：**0**支

赠出金笔：**0**支

荣誉徽章:

精彩图文

第五阶段：存储计算结果到缓冲或内存的阶段。占半个时钟周期；
=> 所以一条指令要占用四个时钟周期；

15、MIPS的虚拟地址内存映射空间：

a 0x0000 0000 ~ 0x7fff ffff
用户级空间，2GB，要经MMU（TLB）地址翻译。kuseg。可以控制要不要经过缓冲。

b 0x8000 0000 ~ 0x9fff ffff
kseg0。这块区域为操作系统内核所占的区域，共512M。使用时，不经过地址翻译，将最高位去掉就线性映射到内存的低512M（不足的就裁剪掉顶部）。但要经过缓冲区过渡。

c 0xa000 0000 ~ 0xbfff ffff
kseg1。这块区域为系统初始化所占区域，共512M。使用时，不经过地址翻译，也不经过缓冲区。将最高3位去掉就线性映射到内存的低512M（不足的就裁剪掉顶部）。

d 0xc000 0000 ~ 0xffff ffff
kseg2。这块区域也为内核级区域。要经过地址翻译。可以控制要不要经过缓冲。

16、MIPS的协处理器
CP0：这是MIPS芯片的配置单元。必不可少，虽然叫做协处理器，但是通常都是做在一块芯片上。绝大部分MIPS功能的配置，缓冲的控制，异常 / 中断的控制，内存管理的控制都在这里。所以是一个完整的系统所必不可少的；

17、MIPS的高速缓冲
MIPS一般有两到三级缓冲，其中第一级缓冲数据和指令分开存储。这样的好处是指令和指令可以同时存取，提高效率。但缺点是提高了复杂度。第二级缓冲和第三级缓冲（如果有的话）就不再分开存放啦。
缓冲的单元叫做缓冲行(cache line)。每一行中，有一个tag，然后后面接的是一些标志位和一些数据。缓冲行按顺序线性排列起来，就组成了整个缓冲。
cache line的索引和存取有一套完整的机制。

18、MIPS的异常机制
精确异常的概念：在运行流程中没有任何多余效应的异常。即当异常发生时，在受害指令之前的指令被完全执行，而受害指令及后面的指令还没开始执行（注：说受害指令及后面的指令还没做任何事情是不对的，实际上受害指令是处于其指令周期的第三阶段刚完成，即ALU阶段刚完成）。精确异常有助于保证软件设计上不受硬件实现的影响。
CP0中的EPC寄存器用于指向异常发生时指令跳转前的执行位置，一般是受害指令地址。当异常时，是返回这个地址继续执行。但如果受害指令在分支延迟槽中，则会硬件自动处理使EPC往回指一条指令，即分支指令。在重新执行分支指令时，分支延迟槽中的指令会被再执行一次。
精确异常的实现对流水线的流畅性是有一定的影响的，如果异常太多，系统执行效率就会受到影响。
* 异常又分常规异常和中断两类。常规异常一般为软件的异常，而中断一般为硬件异常，中断可以是芯片内部，也可以是芯片外部触发产生。
异常发生时，跳转前最后被执行的指令是其MEM阶段刚好被执行完的那条指令。受害指令是其ALU阶段刚好执行完的那条指令。
异常发生时，会跳到异常向量入口中去执行。MIPS的异常向量有点特殊，它一般只含2个或几个中断向量入口，一个入口给一般的异常使用，一个入口给 TLB miss异常使用（这样的话，可以省下计算异常类型的时间。在这种机制帮助下，系统只用13个时钟周期就可以把TLB重填好）。
CP0寄存器中有个模式位，SR(BEV)，只要设置了，就会把异常入口点转移到非缓冲内存地址空间中（kseg1）。
MIPS系统把重启看作一个不可回归的异常来处理。
冷启动：CPU硬件完全被重新配置，软件重新加载；
热启动：软件完全重新初始化；
MIPS对异常的处理的哲学是给异常分配一些类型，然后由软件给它们定义一些优先级，然后由同一个入口进入异常分配程序，在分配程序中根据类型及优先级确定该执行哪个对应的函数。这种机制对两个或几个异常同时出现的情况也是适合的。
下面是当异常发生时MIPS CPU所做的事情：
a 设置EPC指向回归的位置；
b 设置SR(EXL)强迫CPU进入kernel态，并禁止所有中断响应。
c 设置Cause寄存器，以使软件可以得到异常的类型信息；还有其它一些寄存器在某些异常时会被设置；
d CPU开始从异常入口取指令，然后以后的所有事情都交由软件处理了。
k0和k1寄存器用于保存异常处理函数的地址。
异常处理函数执行完成后，会回到异常分配函数那去，在异常分配函数里，有一个eret指令，用于回归原来被中断的程序继续执行；eret指令会原子性地把中断响应打开（置SR(EXL)），并把状态级由kernel转到user级，并返回原地继续执行。

19、中断
MIPS CPU有8个独立的中断位（在Cause寄存器中），其中，6个为外部中断，2个为内部中断（可由软件访问）。一般来说，片上的时钟计数 / 定时器，会连接到一个硬件位上去。
SR(IE)位控制全局中断响应，为0的话，就禁止所有中断；
SR(EXL)和SR(ERL)位（任何一个）如果置1的话，会禁止中断；
SR(IM)有8位，对应8个中断源，要产生中断，还得把这8位中相应的位置1才行；

[查看更多>>](#)

相关博文

[更多>>](#)

推荐博文

- [参加亚投行 日本到底](#)
- [无题的标题，以此悼念周晓辉医生](#)
- [禁欲还是节欲？](#)
- [第1217篇 • 肉身](#)
- [致歉信](#)
- [郭伯雄儿媳郭正钢老婆吴芳芳贪婪](#)
- [第1218篇 • 痰盂——旧物之七](#)
- [“痛经假”荒诞不经，多数女性将](#)
- [官员自杀为何都患“抑郁症”？](#)
- [不要再欺骗善良的民众](#)

[查看更多>>](#)

谁看过这篇博文

 加载中...

中断处理程序也是用通用异常入口。但有些新的CPU有变化。

* 在软件中实现中断优先级的方案

a 给各种中断定优先级；

b CPU在运行时总是处于某个优先级（即定义一个全局变量）；

c 中断发生时，只有等于高于CPU优先级的中断优先级才能执行；（如果CPU优先级处于最低，那么所有的中断都可以执行）；

d 同时有多个中断发生时，优先执行优先级最高的那个中断程序；

20、大小端问题

硬件上也有大端小端问题，比如串口通讯，一个字节一个字节的发，首先是低位先发出去。

还有显卡的显示，比如显示黑白图像，在屏幕上一个点对应显存中的一位，这时，这个位对应关系就是屏幕右上角那个点对应显存第一个字节的7号位，即最高位。第一排第8位点对应第一个字节的0号位。

21、MIPS上的Linux运行情况

用户态和核心态：在用户态，不能随意访问内核代码和数据存放区，只能访问用户态空间和内核允许访问（通过某种机制）的内核页面。也不能执行CP0相关的指令。用户态要执行内核的某些服务，就得用系统调用（system_call），在系统调用的最后，是一个eret指令。

任何时候Linux都有至少一个线程在跑，Linux一般不禁止中断。发生中断时，其环境是从被中断线程借来的。

中断服务程序（ISR）应该短小。

MIPS Linux系统上半地址空间只能用内核特权级访问。内核不通过TLB地址翻译。

所有线程都共用相同的内核地址空间，但只有同一组线程才用同一个用户地址空间（指向同一个mm_struct结构）。

如果物理内存高于512M，那么不能用kseg0和kseg1来映射高于512M的内存部分。只能用kseg2来映射。kseg2要经过TLB。

从某个方面说，内核就是一组供异常处理函数调用的子程序。内核中，线程调度器就是这样一个小的子程序。由各个线程（异常处理程序也可以算作一个特殊的线程，换他书上的说法）调用。

MIPS Linux有异常模式，而x86上没有这个概念。

异常要小心操作。不是仅用软件锁就能解决的。

21、原子操作

MIPS为支持操作系统的原子操作，特地加了一组指令 ll/sc。它们这样来使用：

先写一句

atomic_block:

LL XX1, XXX2

...

sc XX1, XXX2

beq XX1, zero, atomic_block

...

在ll/sc中间写上你要执行的代码体，这样就能保证写入的代码体是原子执行的（不会被抢占的）。

其实，LL/sc两语句自身并不保证原子执行，但他耍了个花招：

用一个临时寄存器XX1，执行LL后，把XXX2中的值载入XX1中，然后会在CPU内部置一个标志位，我们不可见，并保存XXX2的地址，CPU会监视它。在中间的代码体执行的过程中，如果发现XXX2的内容变了（即是别的线程执行了，或是某个中断发生了），就自动把CPU内部那个标志位清0。执行sc 时，把XX1的内容（可能已经是新值了）存入XXX2中，并返回一个值存入XX1中，如果标志位还为1，那么这个返回的值就为1；如果标志位为0，那么这个返回值就为0。为1的话，就表明这对指令中间的代码是一次性执行完成的，而不是中间受到了某些中断，那么原子操作就成功了；为0的话，就表明原子操作没 成功，执行后面beq指令时，就会跳转到ll指令重新执行，直到原子操作成功为止。

所以，我们要注意，插在LL/sc指令中间的代码必须短小。

据经验，一般原子操作的循环不会超过3次。

22、系统调用 syscall

系统调用也通过异常入口进入系统内核，选择8号异常代码处理函数进行处理，进入系统调用分配函数后，还要根据传进来的参数再一次分配到具体的功能函数上去。系统调用传递参数是在寄存器中进行的。

系统调用号存放在v0中，参数存放在a0-a3。如果参数过多，会有另一套机制来处理。系统调用的返回值通常放在v0中。如果系统调用出错，则会在a3中返回一个错误号。

23、异常入口点位于kseg0的底部，是硬件规定的。

24、注意：地址空间的0x0000 0000是不能用的，从0开始的一个或多个页不会被映射。

25、内存分页映射有以下优点：

a 隐藏和保护数据；

b 分配连续的地址给程序；

c 扩展地址空间；

d 按需求载入代码和数据（通过异常方式）；

e 便于重定位；

f 代码和数据在线程中共享，便于交换数据；

所有的线程是平等的，所有的线程都有自己的内存管理结构体；运行于同一地址空间的线程组，共享有大部分这种数据结构。在线程中，保存有本地地址空间已经使用的页面的一个页表，用来记录每个已用的虚页与实际物理页的映射关系；

26、ASID是与虚拟页高位配合使用。用于描述在TLB和Cache中的不同的线程，只有8位，所以最多只能同时运行256个线程。这个数字一般来说是够的。如果超过这个数目了，就要把Cache刷新了重新装入。所以，在这点上，与x86是不同的。

27、MIPS Linux的内存驻留页表结构

用的是两级页表，一个页表目录，一个页表，页表中的每一项是一个 EntryLo0-1。（这与x86方式类似）。而没有用MIPS原生设计的方案。

28、TLB的refill过程—硬件部分

a CPU先产生一个虚拟地址，要到这个地址所对应的物理地址上取数据（或指令）或写数据（或指令）。低13位被分开来。然后高19位成为VPN2，和当前线程的ASID（从EntryHi (ASID)取）一起配合与TLB表中的项进行比较。（在比较过程中，会受到PageMask和C标志位的影响）

b 如果有匹配的项，就选择那个。虚拟地址中的第12位用于选取是用左边的物理地址项还是用右边的物理地址项。

然后就会考察V和D标志位，V标志位表示本页是否有效，D表示本页是否已经脏了（被写过）。

如果V=0，或D=1，就会引发翻译异常，BadVAddr会保存现在处理的这个虚拟地址，EntryHi会填入这个虚拟地址的高位，还有Context中的内容会被重填。

然后就会考察C标志位，如果C=1，就会用缓冲作中转，如果C=0，就不使用缓冲。

这几级考察都通过了之后，就正确地找到了那个对应的物理地址。

c 如果没有匹配的项，就会触发一个TLB refill异常，然后后面就是软件的工作了；

29、TLB的refill过程—软件部分

a 计算这个虚拟地址是不是一个正确的虚拟地址，在内存页表中有没有与它对应的物理地址；如果没有，则调用地址错误处理函数；

b 如果在内存页表中找到了对应的物理地址，就将其载入寄存器；

c 如果TLB已经满了，就用random选取一个项丢弃；

d 复制新的项进TLB。

30、MIPS Linux中标志内存页已经脏了的方式与x86不同。它耍个把戏：

a 当一个可写的页第一次载入内存中时（从磁盘载入？载入的时候就分配一个物理页，同时就分配个对应的虚拟页，并在内存页表中添一个Entry），将其Entry的D标志位清0；

b 然后，当后面有指令要写这个页时，就会触发一个异常（先载入TLB中判断），我们在这个异常处理函数中把内存页表项中的标志位D置1。这样后面的就可以写了。并且，由于这个异常把标志位改了，我们认为这个物理页是脏的了。

c 至于TLB中已经有的那个Entry拷贝还要修改它的D标志位，这样这次写入操作才能继续入下进行。

31、MIPS中的C语言参数传递机制？

32、MIPS中的堆栈结构及在内存中的分布？

指令长度和寄存器个数

MIPS的所有指令都是32位的，指令格式简单。不像x86那样，x86的指令长度不是固定的，以80386为例，其指令长度可从1字节（例如PUSH）到17字节，这样的好处代码密度高，所以MIPS的二进制文件要比x86的大大约20%~30%。而定长指令和格式简单的好处是易于译码和更符合流水线操作，由于指令中指定的寄存器位置是固定的，使得译码过程和读指令的过程可以同时进行，即固定字段译码。

32个通用寄存器，寄存器数量跟编译器的要求有关。寄存器分配在编译优化中是最重要的优化之一（也许是做重要的）。现在的寄存器分配算法都是基于图着色的技术。其基本思想是构造一个图，用以代表分配寄存器的各个方案，然后用此图来分配寄存器。粗略说来就是使用有限的颜色使图中相临的节点着以不同的颜色，图着色问题是个图大小的指数函数，有些启发式算法产生近乎线形时间运行的分配。全局分配中如果有16个通用寄存器用于整型变量，同时另有额外的寄存器用于浮点数，那么图着色会很好的工作。在寄存器数教少时候图着色并不能很好的工作。

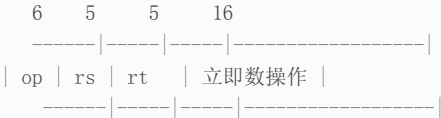
问：既然不能少于16个，那为什么不用64个呢？

答：使用64个或更多寄存器不但需要更大的指令空间来对寄存器编码，还会增加上下文切换的负担。除了那些很大不能感非常复杂的函数，32个寄存器就已足够保存经常使用的数据。使用更多的寄存器并不必要，同时计算机设计有个原则叫“越小越快”，但是也不是说使用31个寄存器会比32个性能更好，32个通用寄存器是流行的做法。

指令格式

所有MIPS指令长度相同，都是32位，但为了让指令的格式刚好合适，于是设计者做了一个折衷：所有指令定长，但是不同的指令有不同的格式。MIPS指令有三种格式：R格式，I格式，J格式。每种格式都由若干字段（field）组成，图示如下：

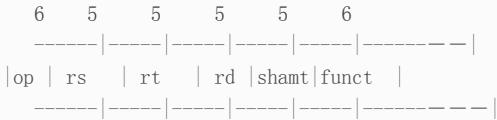
I型指令



加载/存储字节，半字，字，双字

条件分支，跳转，跳转并链接寄存器

R型指令



寄存器-寄存器ALU操作

读写专用寄存器

J型指令



跳转，跳转并链接

陷阱和从异常中返回

各字段含义：

op:指令基本操作，称为操作码。

rs:第一个源操作数寄存器。

rt:第二个源操作数寄存器。

rd:存放操作结果的目的操作数。

shamt:位移量

funct:函数，这个字段选择op操作的某个特定变体。

所有指令都按照着三种类型之一来编码，通用字段在每种格式中的位置都是相同的。

这种定长和简单格式的指令编码很规则，很容易看出其机器码，例如：

add \$t0,\$s0,\$s1

表示\$t0=\$s0+\$s1,即16号寄存器（s0）的内容和17号寄存器(s1)的内容相加，结果放到8号寄存器(t0)。

指令各字段的十进制表示为



op=0和funct=32表示这是加法，16=\$s0表示第一个源操作数(rs)在16号寄存器里，17=\$s1表示第二个源操作数(rt)在17号寄存器里，8=\$t0表示目的操作数(rd)在8号寄存器里。

把各字段写成二进制，为



这就是上述指令的机器码(machine code),可以看出是很有规则性的。

通用寄存器(GPR)

有32个通用寄存器，\$0到\$31：

\$0：即\$zero,该寄存器总是返回零，为0这个有用常数提供了一个简洁的编码形式。MIPS编译器使用slt, beq, bne等指令和由寄存器\$0获得的0来产生所有的比较条件：相等，不等，小于，小于等于，大于，大于等于。还可以用add指令创建move伪指令，即

move \$t0,\$t1

实际为

add \$t0,\$0,\$t1

焦林前辈提到他移植fpc时move指令出错，转而使用add代替的。

使用伪指令可以简化任务，汇编程序提供了比硬件更丰富的指令集。

\$1:即\$at，该寄存器为汇编保留，刚才说到使用伪指令可以简化任务，但是代价就是要为汇编程序保留一个寄存器，就是\$at。

由于I型指令的立即数字段只有16位，在加载大常数时，编译器或汇编程序需要把大常数拆开，然后重新组合到寄存器里。比如加载一个32位立即数需要 lui（装入高位立即数）和addi两条指令。像MIPS程序拆散和重装大常数由汇编程序来完成，汇编程序必需一个临时寄存器来重组大常数，这也是为汇编保留\$at的原因之一。

\$2..\$3: (\$v0-\$v1)用于子程序的非浮点结果或返回值，对于子程序如何传递参数及如何返回，MIPS范围有一套约定，堆栈中少数几个位置处的内容装入CPU寄存器，其相应内存位置保留未做定义，当这两个寄存器不够存放返回值时，编译器通过内存来完成。

\$4..\$7: (\$a0-\$a3)用来传递前四个参数给子程序，不够的用堆栈。a0-a3和v0-v1以及ra一起来支持子程序/过程调用，分别用以传递参数，返回结果和存放返回地址。当需要使用更多的寄存器时，就需要堆栈(stack)了，MIPS编译器总是为参数在堆栈中留有空间以防有参数需要存储。

\$8..\$15: (\$t0-\$t7)临时寄存器，子程序可以使用它们而不用保留。

\$16..\$23: (\$s0-\$s7)保存寄存器，在过程调用过程中需要保留（被调用者保存和恢复，还包括\$fp和\$ra），MIPS提供了临时寄存器和保存寄存器，这样就减少了寄存器溢出(spilling,即将不常用的变量放到存储器的过程)，编译器在编译一个叶(leaf)过程(不调用其它过程的过程)的时候，总是在临时寄存器分配完了才使用需要保存的寄存器。

\$24..\$25: (\$t8-\$t9)同(\$t0-\$t7)

\$26..\$27: (\$k0,\$k1)为操作系统/异常处理保留，至少要预留一个。异常(或中断)是一种不需要在程序中显

示调用的过程。MIPS有个叫异常程序计数器(exception program counter,EPC)的寄存器,属于CP0寄存器,用于保存造成异常的那条指令的地址。查看控制寄存器的唯一方法是把它复制到通用寄存器里,指令`mfcc0(move from system control)`可以将EPC中的地址复制到某个通用寄存器中,通过跳转语句(`jr`),程序可以返回到造成异常的那条指令处继续执行。仔细分析一下会发现个有意思的事情:

为了查看控制寄存器EPC的值并跳转到造成异常的那条指令(使用`jr`),必须把EPC的值到某个通用寄存器里,这样的话,程序返回到中断处时就无法将所有的寄存器恢复原值。如果先恢复所有的寄存器,那么从EPC复制过来的值就会丢失,`jr`就无法返回中断处;如果我们只是恢复除有从EPC复制过来的返回地址外的寄存器,但这意味着程序在异常情况后某个寄存器被无端改变了,这是不行的。为了摆脱这个两难境地,MIPS程序员都必须保留两个寄存器`$k0`和`$k1`,供操作系统使用。发生异常时,这两个寄存器的值不会被恢复,编译器也不使用`k0`和`k1`,异常处理函数可以将返回地址放到这两个中的任何一个,然后使用`jr`跳转到造成异常的指令处继续执行。

\$28: (\$gp) C语言中有两种存储类型,自动型和静态型,自动变量是一个过程中的局部变量。静态变量是进入和退出一个过程时都是存在的。为了简化静态数据的访问,MIPS软件保留了一个寄存器:全局指针 `gp(global pointer, $gp)`,如果没有全局指针,从静态数据去装入数据需要两条指令:一条有编译器和连接器计算的32位地址常量中的有效位;令一条才真正装入数据。全局指针只想静态数据区中的运行时决定的地址,在存取位于`gp`值上下32KB范围内的数据时,只需要一条以`gp`为基指针的指令即可。在编译时,数据须在以`gp`为基指针的64KB范围内。

\$29: (\$sp) MIPS硬件并不直接支持堆栈,例如,它没有x86的SS,SP,BP寄存器,MIPS虽然定义`$29`为栈指针,它还是通用寄存器,只是用于特殊目的而已,你可以把它用于别的目的,但为了使用别人的程序或让别人使用你的程序,还是要遵守这个约定的,但这和硬件没有关系。x86有单独的PUSH和POP指令,而MIPS没有,但这并不影响MIPS使用堆栈。在发生过程调用时,调用者把过程调用后要用的寄存器压入堆栈,被调用者把返回地址寄存器`$ra`和保留寄存器压入堆栈。同时调整堆栈指针,当返回时,从堆栈中恢复寄存器,同时调整堆栈指针。

\$30: (\$fp) GNU MIPS C编译器使用了帧指针(frame pointer),而SGI的C编译器没有使用,而把这个寄存器当作保存寄存器使用(`$s8`),这节省了调用和返回开销,但增加了代码生成的复杂性。

\$31: (\$ra) 存放返回地址,MIPS有个`jal(jump-and-link,跳转并链接)`指令,在跳转到某个地址时,把下一条指令的地址放到`$ra`中。用于支持子程序,例如调用程序把参数放到`$a0~$a3`,然后`jal X`跳到X过程,被调过程完成后把结果放到`$v0, $v1`,然后使用`jr $ra`返回。

在调用时需要保存的寄存器为`$a0~$a3, $s0~$s7, $gp, $sp, $fp, $ra`。

跳转范围

J指令的地址字段为26位,用于跳转目标。指令在内存中以4字节对齐,最低两个有效位不需要存储。在MIPS中,每个地址的最低两位指定了字的一个字节,cache映射的下标是不使用这两位的,这样能表示28位的字节编址,允许的地址空间为256M。PC是32位的,那其它4位从何而来呢? MIPS的跳转指令只替换PC的低28位,而高4位保留原值。因此,加载和链接程序必须避免跨越256MB,在256M的段内,分支跳转地址当作一个绝对地址,和PC无关,如果超过256M(段外跳转)就要用跳转寄存器指令了。

同样,条件分支指令中的16位立即数如果不够用,可以使用PC相对寻址,即用分支指令中的分支地址与(PC+4)的和做分支目标。由于一般的循环和if语句都小于2¹⁶个字(2的16次方),这样的方法是很理想的。

0 zero 永远返回值为0

1 at 用做汇编器的暂时变量

2-3 v0, v1 子函数调用返回结果

4-7 a0-a3 子函数调用的参数

8-15 t0-t7 暂时变量,子函数使用时不需要保存与恢复

24-25 t8-t9

16-25 s0-s7 子函数寄存器变量。子函数必须保存和恢复使用过的变量在函数返回之前,从而调用函数知道这些寄存器的值没有变化。

26,27 k0,k1 通常被中断或异常处理程序使用作为保存一些系统参数

28 gp 全局指针。一些运行系统维护这个指针来更方便的存取“static”和“extern”变量。

29 sp 堆栈指针

30 s8/fp 第9个寄存器变量。子函数可以用来做帧指针

31 ra 子函数的返回地址

这些寄存器的用法都遵循一系列约定。这些约定与硬件确实无关,但如果你想使用别人的代码,编译器和操作系统,你最好是遵循这些约定。

寄存器名约定与使用

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

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

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

*t0-t9: 依照约定,一个子函数可以不用保存并随便的使用这些寄存器。在作表达式计算时,这些寄存器是非

常好的暂时变量。编译器/程序员必须注意的是，当调用一个子函数时，这些寄存器中的值有可能被子函数破坏掉。

*s0-s8: 依照约定，子函数必须保证当函数返回时这些寄存器的内容必须恢复到函数调用以前的值，或者在子函数里不用这些寄存器或把它们保存在堆栈上并在函数退出时恢复。这种约定使得这些寄存器非常适合作为寄存器变量或存放一些在函数调用期间必须保存原来值。

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

* gp: 如果存在一个全局指针，它将指向运行时决定的，你的静态数据(static data) 区域的一个位置。这意味着，利用gp作基指针，在gp指针32K左右的数据存取，系统只需要一条指令就可完成。如果没有全局指针，存取一个静态数据区域的 值需要两条指令：一条是获取有编译器和loader决定好的32位的地址常量。另外一条是对数据的真正存取。为了使用gp，编译器在编译时刻必须知道一个数据是否在gp的64K范围之内。通常这是不可能的，只能靠猜测。一般的做法是把small global data (小的全局数据)放在gp覆盖的范围内(比如一个变量是8字节或更小)，并且让linker报警如果小的全局数据仍然太大从而超过gp作为一个基指针所能存取的范围。

并不是所有的编译和运行系统支持gp的使用。

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

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

如果堆栈的底部在编译时刻不能被决定，你就不能通过sp来存取堆栈变量，因此fp被初始化为一个相对与该函数堆栈的一个常量的位置。这种用法对其他函数是不可见的。

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

子函数如果还要调用其他的子函数，必须保存ra的值，通常通过堆栈。

对于浮点寄存器的用法，也有一个相应的标准的约定。在这里，我们已经介绍了MIPS引入的寄存器指令实例：

1. load/store

```
la $t0, val_1 复制val_1表示的地址到t0寄存器中      注： val_1是个Label
lw $t2, ($t0) t0寄存器中的值作为地址，把这个地址起始的Word 复制到t2 中
lw $t2, 4($t0) t0寄存器中的值作为地址， 把这个地址再加上偏移量4后 所起始的Word 复制到t2 中
sw $t2, ($t0) 把t2寄存器中值（1 Word），存储到t0的值所指向的RAM中
sw $t2, -12($t0) 把t2寄存器中值（1 Word），存储到t0的值再减去偏移量12，所指向的RAM 中
```

2. 算数运算指令

算数运算指令的所有操作数都是寄存器，不能直接使用RAM地址或间接寻址。

操作数的大小都为 Word （4-Byte）

指令格式与实例 注释

```
move $t5, $t1      // $t5 = $t1;
add $t0, $t1,      // $t2 $t0 = $t1 + $t2; 带符号数相加
sub $t0, $t1,      // $t2 $t0 = $t1 - $t2; 带符号数相减
addi $t0, $t1, 5    // $t0 = $t1 + 5;
addu $t0, $t1, $t2  // $t0 = $t1 + $t2; 无符号数相加
subu $t0, $t1, $t2  // $t0 = $t1 - $t2; 无符号数相减
mult $t3, $t4       // $t3 * $t4, 把64-Bits 的积，存储到Lo, Hi中。即： (Hi, Lo) = $t3 * $t4;
div $t5, $t6        // Lo = $t5 / $t6 (Lo为商的整数部分); Hi = $t5 mod $t6 (Hi为余数)
mfhi $t0            // $t0 = Hi
mflo $t1            // $t1 = Lo
```

3. 分支跳转指令

分支指令格式与实例 注释

b target 无条件的分支跳转，将跳转到target 标签处

```
beq $t0, $t1, target // 如果 $t0 == $t1, 则跳转到target 标签处
blt $t0, $t1, target // 如果 $t0 < $t1, 则跳转到target 标签处
ble $t0, $t1, target // 如果 $t0 <= $t1, 则跳转到target 标签处
bgt $t0, $t1, target // 如果 $t0 > $t1, 则跳转到target 标签处
bge $t0, $t1, target // 如果 $t0 >= $t1, 则跳转到target 标签处
```

```
bne $t0, $t1, target // 如果 $t0 != $t1, 则跳转到target 标签处
```

4. 跳转指令

指令格式与实例 注释

```
j target // 无条件的跳转, 将跳转到target 标签处
jr $t3 // 跳转到t3寄存器所指向的地址处 (Jump Register)
```

5. 子函数调用指令

指令格式与实例 注释

- jal sub_routine_label 执行步骤:
- a. 复制当前的PC (Program Counter) 到\$ra寄存器中。 因为当前的PC 值就是子函数执行完毕后的返回地址。
 - b. 程序跳转到子程序标签sub_routine_label处。

注: 子函数的返回, 使用 jr \$ra
如果子函数内又调用了其他的子函数, 那么\$ra的值应该被保存到堆栈中。 因为\$ra的值总是对应着当前执行的子函数的返回地址。

喜欢

0

赠金笔

分享:          

[阅读](#) | [评论](#) | [收藏](#) | [转载原文](#) | [喜欢](#) ▼ | [打印](#) | [举报](#)

前一篇: [Linux内核中的Namespace](#)
后一篇: [MII、GMII、RMII、SGMII、XGMII接口](#)

评论

[发评论]

评论加载中, 请稍候...

发评论

☐  分享到微博  ☐ 匿名评论

验证码: [请点击后输入验证码](#) [收听验证码](#)

发评论

以上网友发言只代表其个人观点, 不代表新浪网的观点或立场。

< 前一篇 [Linux内核中的Namespace](#) [MII、GMII、RMII、SGMII、XGMII接口](#) 后一篇 >

新浪BLOG意见反馈留言板

不良信息举报

电话：4006900000

提示音后按1键（按当地市话标准计费）

欢迎批评指正

新浪简介

About Sina

广告服务

联系我们

招聘信息

网站律师

SINA English

会员注册

产品答疑

Copyright © 1996 - 2014 SINA Corporation, All Rights Reserved

新浪公司 版权所有