仓

我们不生产软件，我们只是代码滴搬运工

MIPS汇编小贴示

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（filed)组成，图示如下：

I型指令

```
    6   5    5    16
     ------|-----|-----|------------------|
  | op | rs | rt  | 立即数操作 |
     ------|-----|-----|------------------|
```

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

```
    6   5    5    5    5    6
     ------|-----|-----|-----|-----|-------——|
  |op | rs  | rt  | rd |shamt|funct  |
     ------|-----|-----|-----|-----|-------———|
```

寄存器-寄存器ALU操作

读写专用寄存器

J型指令

```
    6          26
  ------|------------------------------|
  |op   |     跳转地址          |
     ------|------------------------------|
```

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

```
     ------|-----|-----|-----|-----|------|
  |   0 | 16 | 17 |   8 |   0 |   32 |
     ------|-----|-----|-----|-----|------|
```

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。

昵称：DZQABC
园龄：3年10个月
粉丝：145
关注：2
+加关注

< 2012年8月 >

日	一	二	三	四	五	六
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1
2	3	4	5	6	7	8

搜索

找找看

谷歌搜索

常用链接

我的随笔
我的评论
我的参与
最新评论
我的标签
更多链接

我的标签

js(1)
net(1)
text(1)

随笔档案(784)

2014年8月 (5)
2014年6月 (6)
2014年5月 (1)
2014年4月 (7)
2014年3月 (5)
2014年1月 (2)
2013年12月 (1)
2013年11月 (1)
2013年10月 (8)
2013年9月 (7)
2013年8月 (6)
2013年7月 (7)
2013年6月 (2)
2013年5月 (11)
2013年4月 (7)
2013年3月 (12)
2013年2月 (2)
2013年1月 (8)
2012年12月 (9)
2012年10月 (12)
2012年9月 (4)
2012年8月 (5)
2012年7月 (8)
2012年6月 (7)
2012年5月 (13)

把各字段写成二进制，为

```
------|-----|-----|-----|-----|------|
  |000000|10000|10001|01000|00000|100000|
------|-----|-----|-----|-----|------|
```

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 mfc0(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$29为栈指针，它还是通用寄存器，只是用于特殊目的而已，你可以把它用于别的目的，但为了使用别人的程序或让别人使用你的程序，还是要遵守这个约定的，但这和硬件没有关系。x86有单独的PUSH和POP指令，而MIPS没有，但这并不影响MIPS使用堆栈。在发生过程调用时，调用者把过程调用过后要用的寄存器压入堆栈，被调用者把返回地址寄存器$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编译器使用了侦指针(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2^16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。子函数可以用来做桢指针
31 ra 子函数的返回地□

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於sp的偏移量来存取堆栈上的堆栈变量。详细可参阅10.1节堆栈使用。

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

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

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

子函数如果还要调用其他的子函数，必须保存ra的值，通常通过堆栈。

对於浮点寄存器的用法，也有一个相应的标准的约定。我们将在7.5节。在这里，我们已经介绍了MIPS引入的寄存

1、 MIPS指令集的确很RISC，数据类的仅有load、store和move，当然按操作数的长短分许多lw、lh等等，但实际上就这三个。运算类的也仅仅完成基本功能，也根据操作数长短分了许多子指令。跳转类更少，要么无条件跳转，要么根据操作数跳转。这些指令确实属于最常用的80%的。相比Intel 的LEA等指令，由于个人习惯，很少用，而AAD、AAA等指令，我几乎没用过。

2、MIPS指令较少，但汇编器为了方便使用，定义了许多伪指令，如li、ror等。最终会被扩展成多条实际指令。这样一来，好处就是能省力，但坏处就是对汇编器要求较高，而且对机器指令反汇编后难以还原为伪指令（反汇编器面对lui $at, 0xABCD和ori r, $at, 0xEF00似乎不能自作主张的将其视作li, r, 0xABCDEF00）；反汇编出来的指令条数多，不利于hack（或许又是好事）。

3、MIPS的寻址方式最简单，仅有寄存器加偏移寻址方式（内嵌16位立即数寻址不算在内），这对于饱受Intel的八种寻址方式折磨的人来说是天大的好事。

4、MIPS没有栈操作指令，虽然有约定俗称的$sp。在做递归调用时必须手工管理栈，调用子程序时没有自动压栈的call指令，只能用jal。这对于用惯了intel的PUSH和POP的人又会是一场噩梦。

5、MIPS的内存映射、中断等功能都做到了协处理器0中，浮点运算做到了协处理器1 中。

6、MIPS寄存器非常多，对于表达式求值很有利，不过调度算法就复杂了。而且寄存器虽然有约定俗成的用法，但实际上并没有限制。

7、MIPS指令为定长的，很统一，给我的“感觉”非常好。

最终，个人体会，在MIPS体系下思考又是另一种感觉，由于栈是全手工管理，就不用考虑push、pop是否匹配以及操作数大小，但手工管理栈要求头脑非常清晰；由于寄存器多了，就更多的考虑寄存器调度，如何发挥出所有寄存器的潜力；也不用去费心思选择寻址方式。MIPS在寄存器使用、栈、存储方面提供了更高的灵活性，设计程序可以更加自由，但同时也增大了交流、学习的难度，这点与Intel严格的体系结构完全相反。

从MIPS的特性看来，由于MIPS指令集简单，容易设计和实现，尺寸可以做小，因此MIPS的方向除了嵌入式外，应该是多核心，提高并行度，主要面向并发性高的应用，如服务器。而在桌面应用方面，目前没有x86的优势明显。速度是一方面，MIPS的应用少，指令集太精简、对程序员的友好程度不够好也是一个原因。

分类: GCC/G++/GDB/core dump/调试, MIPS, 转载

« 上一篇：MIPS寄存器介绍
» 下一篇：基于MIPS架构的BackTrace实现

posted @ 2012-08-16 23:59 DZQABC 阅读(1604) 评论(0) 编辑 收藏