# See MIPS Run

**翻译：Alan Yao**



# 10 MIPS 上的 C 语言编程

本章主要讨论用 C 语言建立完整的 MIPS 系统可能需要具备的一些知识，因此，更多时候本章讲述 C 编译器产生的汇编语言代码，而不是 C 语言代码。为避免讨论过于繁琐，而使本章膨胀到一本新书的规模，现假定读者您是第一次向 MIPS 平台移植代码。

一个高效的 C 运行环境依赖于 C 语言程序的寄存器使用约定，这一般由 C 编译器强制规定，因此对于汇编工程师来说，也是需要强制遵守的。参照 2.2.1 部分对寄存器使用的全部约定，本章内容涉及：

- 堆栈、子程序链接、参数传递：关于 MIPS 进程是如何实现的，以及如何为避免不必要工作而支持的各种特性
- 共享库和非共享库：关于在复杂机器上支持共享库 OS 的一点注解
- 介绍编译器的优化：可能对 MIPS 上 C 语言编程造成的影响
- C 语言访问设备的提示：关于如何写绝大多数设备驱动

即使你使用其他的高级语言而非 C 语言，只要你想为 MIPS 编译代码、并与标准库链接，那么本章的大多数内容，还是对你有所帮助的。在这儿，我并没有针对特定编程语言，是因为我对他们了解不够，一直不知道如何恰当的点到为止。

## 10.1 堆栈、子程序链接、参数传递

许多 MIPS 程序使用混合语言编写的——对于嵌入式系统的程序员，这最可能是在 C(或 C++)中加入汇编语言。

一开始，MIPS 社团建立了一套约定，用来规范如何传递参数给函数（在 C 语言中，称为“子例程”或“子程序”）和从函数返回值。这个约定看起来很复杂，其实只是为了逻辑上遵循文档规则，而使文档太过庞大而已。

基本原则是所有参数在堆栈中的一个数据结构中分配空间，只有少数堆栈开始部分的内容可以装入 CPU 寄存器——相应的内存空间将变得是没有定义的。实际上，这意味着对于大多数调用，参数全部传递到寄存器中；然而，堆栈数据结构是理解进程的最好切入口。

自从 1995 年 Silicon Graphics 开始，已经为了提高性能而对调用约定作了修改。并对这些修改作如下命名：

- o32：传统的 MIPS 约定（o 是 old 简写）；详细说明如下。这个约定（不包括 SGI 为支持共享库而添加的一些特性）目前还是嵌入式工具相当常用的；不过过不了多久，两个最新的约定将会被其他工具作为可选项加以支持。
- n64：针对 64 程序的约定。SGI 的 64 模式意味着指针和 C 语言的 long 整数类型都是 64 位的。对于嵌入式应用程序，更长的指针意味着越界，这样会产生疑惑：这个约定现在是否用在了工作站环境上？不过，n64 改变了使用寄存器的约定和参数传递规则，因为 n64 将更多的参数放进了寄存器，从而提高了性能。
- n32：在参数传递上采用了 n64 的规则，不过指针和 C 语言的 long 整数类型都是 32 位的。然而，SGI 和其他编译器支持扩展的 long long 整数类型，从而实现硬件支持的 64 位整数。这个编译模式在嵌入式系统变得很流行

这里先描述 o32 标准，然后指出 n32 和 n64 的差异部分（10.8 节）。

这本书出版时，还有其他有争议的标准，不过大多数和 MIPS EABI 相类似。所有的 MIPS EABI 项目的目的是要产生一个范围更广的标准，以便嵌入式工具能相互工作的更好。这是个绝好的主意，不过这个新的调用约定是从类似 SGI n32 的私人项目继承来的（虽然简单，但没有很好的兼容性）。我们虽然很希望这能产生很好的结果，但是目前在嵌入式应用中合理使用 o32 编译模式，也不会失去什么。

## 10.2 堆栈参数结构

从这节开始，将陆续介绍 SGI 称为 o32 的原始 MIPS 约定。并在 10.8 节才开始明确介绍新约定变化。

MIPS 硬件不直接支持堆栈，但是调用约定需要。堆栈是向下延伸的，当前堆栈的底保存在寄存器 sp($29)中。任何提供保护和安全的 OS 都不支持用户堆栈，而且除非在函数调用的地方，sp 的值没有价值。但是约定还是在函数使用的堆栈的最下方保留了 sp。

在函数调用的地方，sp 必须是 8 字节对齐的（对于 32 位 MIPS 硬件，不是必须的；但是对于 64 位 CPU，是必须的）。子程序总是将堆栈指针调整为 8 的倍数，然后填到 sp 中。(注：SGI 的 n32 和 n64 标准调用的堆栈都是以 16 字节对齐的方式维护的)

在 MIPS 标准中，为了调用子程序，调用者在堆栈中建立一个数据结构来保存参数，并将 sp 指向这个数据结构。第一个参数（在 C 程序中位于最左的）在内存中是处在最下方。每个参数至少占据一个 word（32 位）；64 位的值，比如浮点 double 类型和(对于某些 CPU)64 位整数，必须是 8 字节对齐的（就好像是个包含 64 位“纯量场 scalar field”的数据结构）。

参数结构就像一个 C 的 struct，但是会有更多的规则。首先，为任何调用分配一个至少 16 字节的参数空间，即使没有参数。其次，char 和 short 等比 word 类型短的数据类型，以一个 int 类型（32 位）传递。不过，这种处理方式不能用于 struct 内部。

## 10.3 使用寄存器传递参数

任何位于参数结构开头 16 字节内的参数都被传递到寄存器中，调用者可以不明确定义参数结构中的这 16 字节。存在于堆栈中的参数结构必须保存下来；如果必要，被调用的函数有权把寄存器中参数的值存回到内存中（可能是在 C 中，参数是一个指向变量的指针）。

除非调用者确认数据存在浮点寄存器(FP)中更适合，否则，四个寄存器中参数分别存在 a0-a3($4-$7)中。

决定何时以及如何使用 FP 寄存器的标准是很特别的。就风格的 C 没有内建机制检查调用者和被调用者协调函数每个参数的类型。为了帮助程序员避免混乱，调用者将参数转换成固定类型，整数用 int，浮点数用 double。对于分不清整数和浮点数的程序员，实在是没有办法了，不过，这样至少减少了一些混乱。

现代的 C 编译器使用函数原型，定义所有参数类型，并能让所有调用者看到。即使这样，还会有程序的参数类型在编译的时候是不确定的，比如著名的 printf()；printf()是在运行时才确定自己参数的数量和类型。

MIPS 制定了如下的规则。

除非第一个参数是浮点类型，否则不能将后续参数传递到 FP 寄存器中。这样可以保证 printf()等传统函数能正常工作：printf()第一个参数是指针，所以所有后续参数都分配到整数寄存器中，printf()因此能够在参数数据中找到所有参数（不考虑参数类型）。这个规定不会使普通的数学函数效率下降，因为这些函数大多数使用浮点参数。

如果第一个参数是浮点类型，那将会传递到 FP 寄存器中，这样参数结构前 16 字节的其他后续浮点参数也会被传递到 FP 寄存器中。两个 double 占据 16 字节，因此只有两个 FP 寄存器（fa0-fa1 或$f12-$14）用于参数定义。显然不会有人认为函数明确定义大量的单精度参数是常见的事，而非要定义一个新的规则来处理。

另外一个比较特别的是，定义一个函数，它的返回结构大于正常使用的两个寄存器，那么，返回值约定要求产生一个指针作为参数，指向这个结构，并放在其他普通参数前传入（详见 10.7 节）。

如果需要写一个调用约定不是很简单和显而易见的汇编程序，可以先用 C 编写程序，并用编译器带上-S 选项产生汇编文件，以此作为模板。

## 10.4 C 库范例

这里举个例子：

**thesame = strcmp ( “bear”, “bearer”, 4 );**

在 figure 10.1 中，画出了分别参数结构和寄存器内容，这里没有参数数据是放在内存中的，在后续部分会举出那方面的实例。

| 堆栈位置 | 内容 | 寄存器 | 内容 |
|---|---|---|---|
| sp+0 | undefined | a0 | address of “bear” |

| | | | |
|---|---|---|---|
| sp+4 | undefined | a1 | address of “bearer” |
| sp+8 | undefined | a2 | 4 |
| sp+12 | undefined | a3 | undefined |

FIGURE 10.1 **参数结构，三个非浮点操作符**

**参数数据不足** 16 **字节，所以参数都存在寄存器中。**

**看起来，决定将三个参数放在普通寄存器中式荒谬的复杂方法，但是在看看数学库中一些巧妙方法：**

```
double  ldexp  ( double ,  int  ) ;
y = ldexp ( x ,  23  ) ; /*  y  =  x  * ( 2  ** 23 ) */
```

Figure 10.2 **显示了相应的参数结构和寄存器值。**

| **堆栈位置** | **内容** | **寄存器** | **内容** |
|---|---|---|---|
| sp+0 | undefined | $f12 | (double) x |
| sp+4 | | $f13 | |
| sp+8 | undefined | a2 | 23 |
| sp+12 | undefined | a3 | undefined |

FIGURE 10.2 **参数传递：一个浮点参数**

## 10.5 一个特殊的例子：传递数据结构

C **允许使用数据结构类型作为参数（实际上传递的是数据结构的指针，不过** C **语言同时支持这两种方式）。为了适应** MIPS **的规则，传递的数据结构参数只能是参数结构的一部分。在一个** C **的数据结构中，**byte **和** halfword **域会共用一个** word **的位置存放在内存中，因此当通过寄存器传递堆栈中参数结构的参数时，也必须这样处理。**

**因此，如果是这样：**

```
struct thing {
    char    letter ;
    short   count ;
    int     value ;
```

**} = { “z” , 46 , 100000 } ;**

**( void ) processthing ( thing ) ;**

**那么， 将会产生** Figure 10.3 **显示的参数结构。**

| 堆栈位置 | 内容 | 寄存器 | 内容 |
|---|---|---|---|
| sp+0 | undefined | a0 | ”z” \| x \| 46 |
| sp+4 | undefined | a1 | 100000 |

FIGURE 10.3 **传递数据结构作为参数**

**注意，因为** MIPS **的** C **数据结构以域分布的，内存中的顺序和定义数据结构时的顺序是一致的（不过填充时，要尽量满足对齐原则），这些域存放在寄存器中的位置是遵循** load/store **指令的字节顺序，因** CPU **的字节序而异。**Figure 10.3 **是针对大字节序** CPU **的，因此数据结构中的** char **值是在放置参数的寄存器最高** 8 **位，并和** short **对齐。**

**如果真想传递数据结构作为参数，而且一定包含短于** short **的数据类型，那就应该测试一下这种情况，看看编译器是否处理正确。**

## 10.6 传递不定数量的参数

**对于传递的参数数量和类型在运行时才能确定的函数，约定对他们的限制是很严格的。考虑这样的例子：**

**printf ( “ length = %f , width = %d \ n”, 1.414 , 1.0 , 12 ) ;**

**根据前面的规定，参数结构和寄存器的内容如** Figure 10.4 **所示：**

| 堆栈位置 | 内容 | 寄存器 | 内容 |
|---|---|---|---|
| sp+0 | undefined | a0 | format pointer |
| sp+4 | undefined | a1 | undefined |
| sp+8 | undefined | a2 | (double) 1.414 |
| sp+12 | | a3 | |
| sp+16 | (double) 1.0 | | |
| sp+20 | | | |

sp+24 | undefined |
---

FIGURE 10.4 printf()参数传递

这里有两件事需要注意。首先，sp+4 中存放的内容需要和 double 类型值对齐（在 C 规则中，浮点参数需要以 double 类型传递，除非通过类型说明和函数原型做出明确的定义）。注意，8 字节内容会导致浪费掉一个标准参数寄存器（译者注：比如 Figure 10.4 中的 a1）。

第二，第一个参数不是浮点参数，根据前述规则，不能给参数分配 FP 寄存器。因此，第二参数的数据（和内存中的一致）只能存放在 a2-a3。

这看似简单，但却是非常实用的。

printf()子程序定义用到的宏是在 stdarg.h 中定义。stdarg.h 提供可移植的接口，用于寄存器和堆栈关于对不定数量和不定类型的操作符进行的操作。printf()解析所有参数，是通过获取第一参数的地址和第二个参数、并在内存中向上寻找参数结构来实现的。

未达到这样的目的，需要 C 编译器将 printf()的 a0-a3 寄存器放到参数结构的范围内。一些编译器会检查是否获得参数的地址，并满足这样的隐性要求；ANSI C 编译器函数定义中使用“…”来做出这样的提示；其他编译器可能使用讨厌的“pragma”来提示，幸运的是，不久 stdarg.h 的宏中就不会有它了。

现在可以看出将 double 类型值放入整数寄存器的必要性了；这样，stdarg 和编译器只要将 a0-a3 存放到参数结构的前 16 字节，而不用考虑参数的数量和类型。

## 10.7 函数的返回值

函数返回的整数或指针类型，通常是放在寄存器 v0($2)。虽然大多数编译器都不会用到寄存器 v1($3)，不过再 MIPS/SGI 定义的预定中，寄存器 v1($3)是保留不用的。不过在 32 位模式当中返回 64 为非浮点类型的数据时，会用到这个寄存器。有些编译器定义 64 位数据类型(通常是 long long)，这时也会使用 v1 寄存器以返回一个 64 位的数据结构值，而避免 32 位的限制。

所有浮点类型的值都放在寄存器$f0 中返回（在 32 位的 CPU 中，双精度的值也默认使用寄存器$f0）。

如果 C 中的函数返回的数据结构太大，不能通过 v0-v1 返回，就需要作额外处理。这时调用者要在堆栈中为这个数据结构预留一些空间，并用一个指针指向这个空间；被调用的函数将返回值拷贝到这个地方。一般的参数规则要求在函数调用时，将这个指针放在寄存器 a0 传入中，放在 v0 中返回。

## 10.8 扩展的寄存器使用标准：SGI n32 和 n64

在调用约定和寄存器使用上，n32 和 n64 的 ABI 是一致的（注：不同的是，在 n64 中，long 和指针类型都是 64 位的，而在 n32 中，只有 long long 类型是 64 位的）。

尽管保持寄存器使用约定的一致性时非常有益的事，但是 o32 和 n32/n64 有很大的区别，用不同的方法编译函数，并且不能成功链接到一起。归纳一下 n32/n64 的新规则，主要有以下几点：

- 提供至多 8 个参数通过寄存器来传递。
- 参数和用于参数传递的寄存器都是 64 位的，长度短于整数的类型，都将以 64 位的

形式操作

- 不需要调用者非寄存器中的参数分配堆栈空间
- 尽可能的通过寄存器传递数据结构和数组（传递方法和旧标准相似）
- 前 8 个参数中的浮点值都通过 FP 寄存器传送。实际上，除了在 union 类型和类似 printf()这样参数不确定的函数中，数据结构和数组中和 double 等长的数据类型都将使用 FP 寄存器

允许传递数据结构和数组后，情况变得复杂了，不过即使现在没有堆栈空间可以保留，寄存器的使用相对于复杂参数结构，还是清晰可见的。

n32/n64 废除了 o32 中一个约定，那就是在类似 printf()函数的参数不确定时，要求第一参数必须是非浮点的约定，以便区别普通浮点参数的使用情况。新约定要求调用者和被调用函数在编译时有明确的函数数量和类型，这是通过函数原型来实现的。

n32/n64 有一套不同的寄存器使用约定，Table 10.1 罗列出了于 o32 的区别。唯一的区别在于：单纯的用于临时存储的四个寄存器现在可以传递第 5-8 个参数。对临时寄存器改变（译者注：这里是指 o32 中的 t0-t3 临时寄存器改成 n32/n64 中的 a4-a7 参数寄存器），显然是没有必要的，我对他们的这个做法有点迷惑不解。

TABLE 10.1 新 SGI 工具中整型寄存器使用解决方法

| 寄存器号 | 名称 | 用途 |
|---|---|---|
| $0 | zero | 始终为 0 |
| $1 | at | 汇编编译器使用的临时寄存器 |
| $2,$3 | v0,v1 | 函数返回值 |
| $4-$7 | a0-a3 | 函数参数 |
| $8-$11 | t0-t3 | 临时寄存器 |
| $12-$15 | t4-t7 | |
| $24,$25 | t8,t9 | |
| $16-$23 | s0-s7 | 保全寄存器（saved register） |
| $26,$27 | k0,k1 | 留给 interrupt/trap 的处理程序使用 |
| $28 | gp | 全局指针（Global pointer） |
| $29 | sp | 堆栈指针 |
| $30 | s8/fp | 需要时作为帧指针(Frame pointer)；否则作为附加的保全寄存器 |
| $31 | ra | 子程序返回的地址 |

| 032 | | n32/n64 | |
|---|---|---|---|
| 名称 | 用途 | 名称 | 用途 |
| $8-$11 | t0-t3 | a4-a7 | 参数 |
| $12-$15 | t4-t7 | t0-t3 | 临时寄存器 |
| $24,$25 | t8,t9 | t8,t9 | |

表面上，这样可以避免编译器产生的代码丢失曾经存储在四个临时寄存器中的值，但实际上，大多数时候编译器可以使用所有参数寄存器和 v0-v1 寄存器作为临时存储器，达到同样的效果。而且 n32/n64 这一修改，没有影响“保全(saved)”的寄存器(可以假定能从子程序带回值得寄存器)。（注：事实并非如此。在 SGI 计算机上，函数使用 gp 寄存器帮助实现代码的位置独立性，详见 10.11.2。在 o32 中，每个函数以类似的方法使用 gp，这意味着每个函数调用后，不得不恢复 gp 寄存器。在 n32/n64 中，gp 被定义成“保留未使用” 寄存器。

在大多数嵌入式系统中，gp 是常量，因此，上述区别只是理论上的。）

浮点寄存器的约定（见 Table 10.2）是修改比较大的；这没什么可以大惊小怪的，因为 MIPS III CPU 有 16 个额外的 FP 寄存器。在旧的系统体系结构中，偶数寄存器的使用通常会附带使用奇数寄存器（译者注：这样，寄存器的长度就被翻倍使用了，即 2n+0 处的 8 字节内容就放在了 2n+0 和 2n+1 处的两个 4 字节空间里）。（注：MIPS III CPU 有个模式切换，使 FP 的使用和早期的 32 位 CPU 兼容；n32/n64 假定 CPU 不支持这种兼容。）本来 SGI 可以添加一些新的寄存器保持兼容性，但是他们却偏偏要修改大多数现有规则，从新开始。

TABLE 10.2 o32 和 n32/n64 约定中 FP 寄存器使用

| 寄存器号 | 在 032 中的用途 |
|---|---|
| $f0,$f2 | 返回值；fv1 只用于复杂的数据类型，不能在 C 中使用 |
| $f4,$f6,$f8,$f10 | 临时寄存器，函数中用于不需要保留的值 |
| $f12,$f14 | 参数寄存器 |
| $f16,$f18 | 临时寄存器 |
| $f20,$f22, $f24<br>$f26,$f28, $f30 | 保全寄存器。为了保证函数调用过程中，这些寄存器中的值长期有效，在对这些寄存器写操作时，必须保存和恢复这些寄存器的值 |

| 寄存器号 | 在 n32 中的用途 | 在 n64 中的用途 |
|---|---|---|
| $f0,$f2 | 返回值；$f2 只用于处理 Fortran 复数返回值正好是两个浮点值， | |
| $f1,$f3,$f4-$f10 | 临时寄存器 | |
| $f12-$f19 | 参数寄存器 | |
| $f20-$f23 | 偶数($f20-$f30)是临时寄存器 | 临时寄存器 |
| $f24-$f31 | 奇数（$f21-$f31）是保全寄存器 | 保全寄存器 |

除了可以通过寄存器传递更多参数，n32/n64 没有依赖是否第一参数采取浮点类型而制定新的规则。实际上，参数根据在参数列表中的位置决定如何分配给寄存器。再看看前面的一个例子：

```
double ldexp ( double , int );
y = ldexp ( x , 23 ); /* y = x * (2**23) */
```

Figure 10.5 显示了 n32/n64 中寄存器和堆栈结构的值。

| 堆栈位置 | 内容 | 寄存器 | 内容 |
|---|---|---|---|
| sp+0 | undefined | $f12 | (double) x |
| sp+4 | | $f13 | |
| sp+8 | undefined | a1 | 23 |

FIGURE 10.5 n32/n64 参数传递：一个浮点参数

尽管 n32/n64 能处理各种浮点和其他类型数据的混合情况，但还是把前 8 个参数中任何

double 类型放在 FP 寄存器中。不过，这不包含 union 类型和参数函数不定的情况（这些情况下，也许不是正真的 double 类型）。注意这是基于函数原型机制，如果没有函数原型机制，那就会有问题。SGI 链接器通常会作检测，并给警告信息。

## 10.9 堆栈布局、堆栈帧、辅助调试器

Figure 10.6 给出了 MIPS 函数堆栈帧（stack frame）概况图。（现在起，堆栈恢复以前那种增长模式，高内存空间在上面）。传统 MIPS 调用约定要求函数参数前 4 个 word 要给于保留，而在新的调用约定中，只是在需要时分配空间。

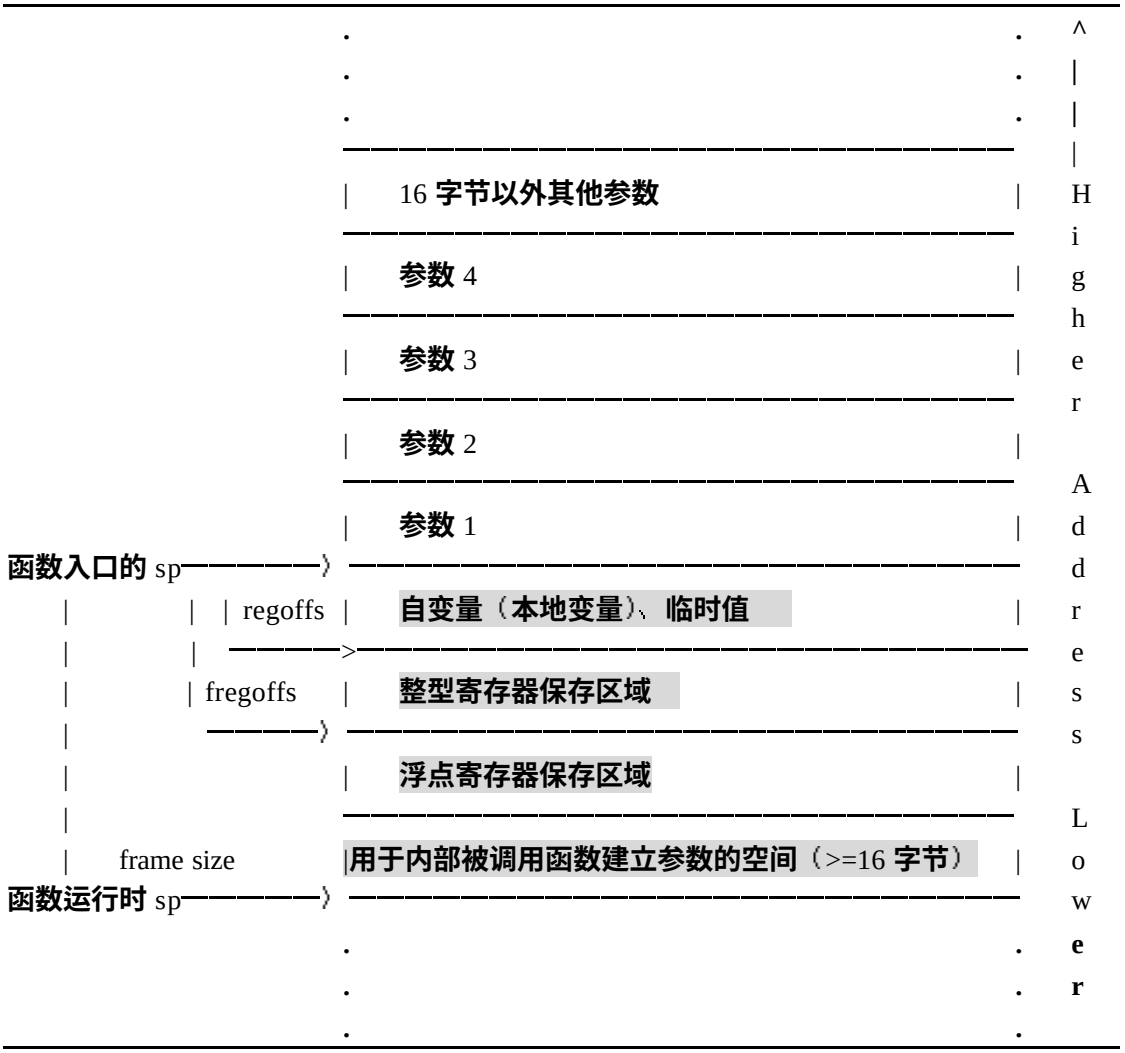



FIGURE 10.6 一个 nonleaf 函数的堆栈帧

（译者注：nonleaf 函数是指内部会调用其他函数的函数，而其他函数就是深层调用 nest call）

显灰的部分是被调用函数自己需要的堆栈空间；上面的白底区域属于调用者。堆栈帧中所有显灰的部分都是可选的，有些函数一个也不需要。非常简单的函数是不需要对堆栈作什么处理的。不过在后续部分，会举例说明其中的一部分。

除了参数（布局需要和调用者一致），堆栈结构对于函数来说，是私有的。需要这种标准布局的唯一理由是用于调试和诊断工具，它们常常需要能够操纵堆栈。如果调试过程中，

中断一个正在运行的函数，通常希望能够在堆栈中回溯，显示运行到当前断点处所调用的函数列表和这些函数的参数列表。而且，希望能够将调试器的上下文回复到堆栈中某个新位置，察看那时某个变量的值；即使维护寄存器中变量数值的代码很少，经过优化的编译器还是能够通过这种机制实现这样的目的。

为了做出正确的分析，调试器必须知道标准的堆栈分布，从而得到必要信息，能够知道每个堆栈帧内容的尺寸和内部布局。如果一个函数在前面的堆栈中用 s0 保存值，以便今后使用，那么调试器需要知道到哪里去找这个保存的值。

在 CISC 体系结构中，经常会有一个复杂的函数调用指令来维系类似于 Figure 10.6 中的堆栈帧，不过还需要一个附加的帧指针寄存器来存储函数入口的 sp。在这样的 CPU 中，调用者的帧指针被保存在大家共知的堆栈位置，以便允许调试器可以忽略堆栈的内容，而只是分析一个简单的链接列表，就能达到目的。但是在 MIPS CPU 中，不需要在运行时间做这些额外工作；大多数时候，编译器知道在函数入口降低堆栈指针和在函数返回时增加堆栈指针。

那么，在这个最小限度的 MIPS 堆栈帧中，调试器如何知道在何处找到曾经保存的值呢？一些调试器做的非常漂亮，甚至通过解析函数前几个指令，就能发现堆栈帧的尺寸和返回地址存放的位置。不过大多数工具根据汇编编译器的指示，在目标代码中适当的放些堆栈帧的信息。

和汇编编译器相关的这种指示很多，因此需要定义一些宏，对这些指示的支持做开关切换，这样免除记忆具体细节差异的痛苦，并在必要时用于和其他工具间的切换。现在大多数工具都已经使用这些宏了；下面的例子使用 LEAF 和 NESTED 宏来实现这样的目的。

对 SGI 约定做全面的描述，是没有什么必要的。下面的例子（使用前面推荐的启始宏和结束宏）是和旧版本 SGI 工具兼容，因此能够被大多数嵌入式工具兼容。

关键的指示.frame 和.mask，可以在 9.5 节找到详细的说明。

为了说明各种问题，我们将函数分成三种类型，并分别加以讲解。

## 10.9.1 leaf 函数

内部不调用其他函数的函数，成为 leaf 函数。这些函数不用担心建立参数结构和安全维护没有保存的寄存器 t0-t7、a0-a3、v0、v1 数据，可以用堆栈存储数据，并在寄存器 ra 中存放返回地址，在函数运行结束后，通过这个地址直接返回。（注：将返回地址存储在别的地方，可能表现得更好，但是调试器可能找不到。）

有时为了优化或实现 C 无法实现的功能，通常需要用汇编写一些函数，这些函数一般会是 leaf 函数；许多这些函数根本就不使用堆栈空间。声明这样的函数比较简单，比如：

```
#include <mips/asm.h>
#include <mips/regdef.h>

LEAF    ( myleaf )
------
< your code gose here >
------
j    ra
END     ( myleaf )
```

大多数工具可以在汇编前通过 C 宏预处理器将汇编源码传入，——unix 风格的工具将根据文件的扩展名做出判断。mips/asm.h 和 mips/regdef.h 在定义全局函数和数据的时候非常有用的宏（比如前面的 LEAF 和 END）；同时可以允许直接使用寄存器名字，比如用 a0 来

代表$4。如果使用旧的 MIPS 或 SGI 工具。上面的那段代码将会扩展成:

```
.globl    myleaf
.ent      myleaf,0
------
< your code gose here >
------
j         $31
.end      myleaf
```

其他工具使用的宏可能有所不同，不过实现的功能是大同小异。

## 10.9.2 nonleaf 函数

内部调用其他函数的函数，称为 nonleaf 函数。通常这些函数需要在开始处，重新设定 sp 寄存器指向所有被其调用函数的参数结构之后的位置，同时保存这些被调用函数所用到的 s0-s8 寄存器最新值。必须保存 ra 寄存器、自变量（也就是堆栈本地变量）和函数在运行后需要保存值的其他寄存器的堆栈位置。（如果参数寄存器 a0-a3 的只需要保存，可以存放到参数结构的标准位置）。

注意，只设定一次 sp（在函数的入口处），所有数据在堆栈中的位置，都通过 sp 加上固定的偏移来获取。

为了解释这点，将通过下面这个 nonleaf 函数说明。（联系 Figure 10.6 的堆栈帧的分布图理解）。

```
#include  <mips/asm.h>
#include       <mips/regdef.h>

#
#myfunc (arg1, arg2, arg3, arg4 ,arg5)
#

#framesize = locals + regsave(ra,s0) + pad  + fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 8 + 4 +8 + (5*4) + 4

NESTED( myfunc , myfunc_frmsz , ra)
     subu          sp,myfunc_frmsz
     .mask    0x80010000 , -4
     sw       ra , myfunc_frmsz-8 (sp)
     sw       s0 , myfunc_frmsz-12 (sp)
     .fmask   0x00300000 , -16
     s.d      $f20 , myfunc_frmsz – 24 (sp)
     ------
     < your code goes here , e.g.>
     # local = otherfunc(arg5,arg2,arg3,arg4,arg1)
     sw       a0 , 16 (sp)                    # arg5(out) = arg1(in)
     lw       a0 , myfunc_frmsz + 16 (sp)     # arg1(out) = arg5(in)
     jal      otherfunc
```

```
sw      v0 , myfunc_frmsz – 4    (sp)
.....
l.d     $f20 , myfunc_frmsz –  24   (sp)
lw      s0 , myfunc_frmsz –  12   (sp)
lw      ra , myfunc_frmsz –  8   (sp)
addu        sp , myfunc_frmsz
jr      ra
END (myfunc)
```

上面代码开始处显示函数 myfunc 有五个参数：在函数入口处，前四个参数放在 a0-a3 中，第五个参数放在 sp+16 处。接下来的这些代码：

```
#framesize = locals + regsave(ra,s0) + pad  + fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 8 + 4 +8 + (5*4) + 4
```

堆栈帧的尺寸计算方式如下：

- local（4 字节）：在堆栈中而不是寄存器保留一个本地变量，可能是需要将这个变量地址传递给其他函数。
- regsave（8 字节）：用来在寄存器 ra 中存放返回地址，因为这个函数会在内部调用其他函数，这可能会用到被调用函数的 s0 保全寄存器
- pad（4 字节）：因为后续的 fregsave 是双精度浮点寄存器，根据规则需要 8 字节对齐，因此这里填补一个 word
- fregsave（8 字节）：用$f20 作为被调用函数的保全浮点寄存器。
- args（20 字节）：内部被调用函数需要五个参数。不过即使这个 nested 函数没有参数，但如果其内部被调用函数是 nested 函数，则这部分的长度不得低于 16 字节。
- pad（4 字节）：堆栈指针也不许是 8 字节对齐的，因此这里也需要一个 word 填补。

接下来的代码：

```
NESTED( myfunc , myfunc_frmsz , ra)
    subu        sp,myfunc_frmsz
```

在 MIPS 公司的工具中，这段代码扩展成：

```
.globl  myfunc
.ent    myfunc , 0
.frame  $29 , myfunc_frmsz , $0
subu    $29 , myfunc_frmsz
```

这里声明了 myfunc 函数的入口，并使它变成全局函数而被访问。.frame 告诉调试器这个函数建立的堆栈的尺寸，subu 指令建立堆栈。

接下来的代码：

```
.mask   0x80010000 , -4
sw      ra , myfunc_frmsz-8 (sp)
sw      s0 , myfunc_frmsz-12 (sp)
```

函数必须保存返回地址和这个堆栈帧中所有被调用函数的保全寄存器。.mask 告诉调试器需要保存的寄存器（$31 和$16，也就是 ra 和 s0）和这些寄存器保存区域顶部相对于堆栈帧顶部的偏移：也就是 Figure 10.6 中的 regoffs。sw 指令保存这些寄存器的值；寄存器号大的存放在堆栈中的位置偏上（也就是寄存器位置排列沿内存地址增加方向的）。接下来的代码：

```
.fmask  0x00300000 , -16
s.d     $f20 , myfunc_frmsz – 24 (sp)
```

对被调函数的保全浮点寄存器$f20 和（隐含的）$f21 做同样的处理。.fmask 的偏移对应于 Figure 10.6 中的 fregoffs（也就是 自变量区域+整形寄存器保存区域+为了对齐而填补的 word）。接下来的代码：

```
# local = otherfunc(arg5,arg2,arg3,arg4,arg1)
sw      a0 , 16 (sp)                    # arg5(out) = arg1(in)
lw      a0 , myfunc_frmsz + 16 (sp)     # arg1(out) = arg5(in)
jal     otherfunc
```

开始调用 otherfunc 函数，这个函数的参数 2-4 是和 myfunc 函数一样的，因此不需要移动，就可以直接传给过来。而参数 1 和参数 5 需要调换一下：将 arg1（在 a0 寄存器中）拷贝到输出参数区域（sp+16）作为被调用函数的 arg5，将 arg5（在 sp+16 处）拷贝到输出参数 1(放在寄存器 a0 中)。

接下来的代码：

```
sw      v0 , myfunc_frmsz – 4   (sp)
```

otherfunc 函数的返回值存放在自变量（本地变量）中；位于堆栈帧的开始 4 字节。

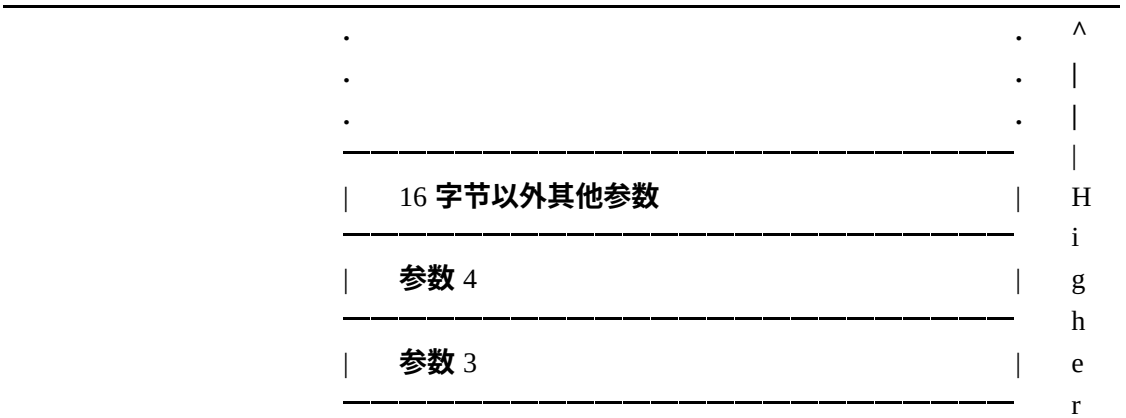
最后：

```
l.d     $f20 , myfunc_frmsz – 24   (sp)
lw      s0 , myfunc_frmsz – 12   (sp)
lw      ra , myfunc_frmsz – 8   (sp)
addu    sp , myfunc_frmsz
jr      ra
END (myfunc)
```

是做一些函数结束时的处理工作：恢复浮点寄存器、整形寄存器和存放返回地址的寄存器；弹出堆栈帧，并返回。

## 10.9.3 复杂堆栈请求的堆栈帧指针

在前面的堆栈帧描述中，编译器能够管理只需保存 sp 寄存器的堆栈。对于熟悉其他体系结构的程序员来说，经常会使用两个寄存器来管理堆栈，sp 指向堆栈底端，堆栈帧指针指向函数在入口处建立的数据结构。然而，只要编译器能够在函数入口处分配函数所需的堆栈空间，那就能在入口处增加 sp，在函数运行期间，使它指向一个固定的堆栈偏移地址。如果这样，本地堆栈帧中的所有内容，在编译时就确定了相对于 sp 的偏移量，因此不需要额外的堆栈帧指针。但是有时候，在运行过程中堆栈指针会出现混乱：Figure 10.7 显示了 MIPS 是如何针对这种情况分配堆栈帧指针的。



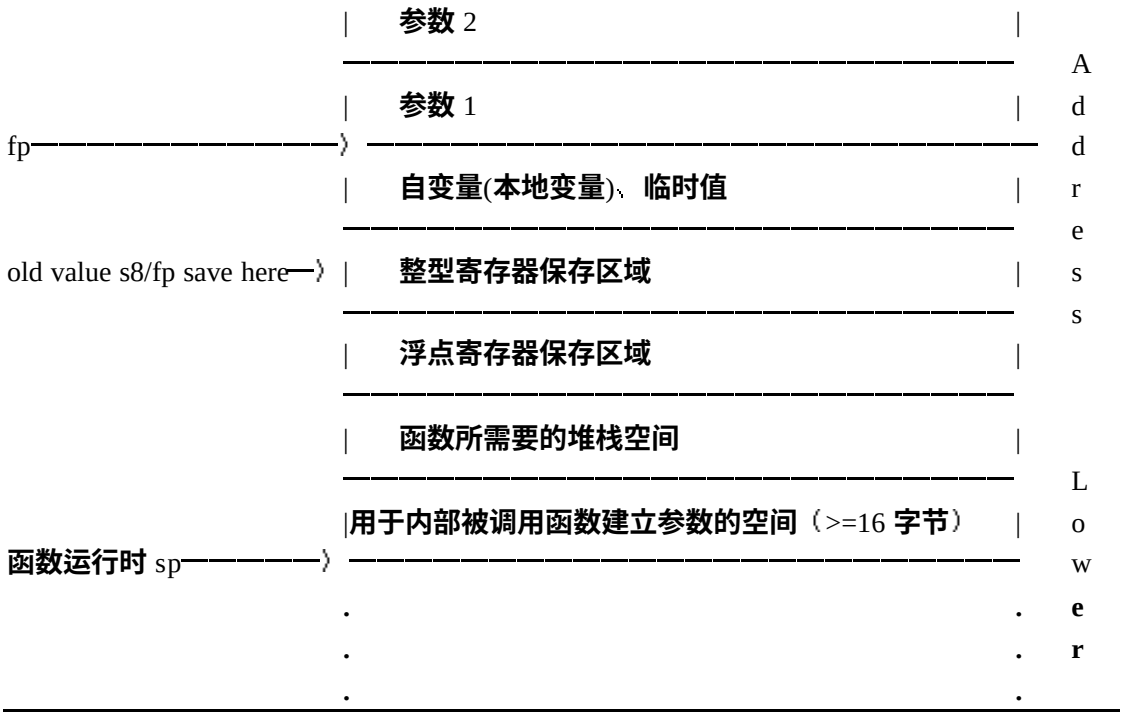



FIGURE 10.7　堆栈帧使用单独的堆栈帧指针寄存器

什么情况下会用到这种机制呢？在一些编程语言中，甚至一些 C 的扩展语言中，会创建在运行时才确定尺寸的动态变量。而且很多 C 编译器使用非常实用的 alloca()内建函数，在运行时按要求分配堆栈空间。这时，函数的入口处，需要使用额外的 s8 寄存器（也就是 fp）来保存此时 sp 值。

既然 fp 寄存器（也就是 s8）是个保全寄存器，函数入口处也必须保存它的原有值，方法和在子程序中作为一个变量保存 s8 一样。在编译时附带了堆栈帧指针的函数里，所有对堆栈帧内部的访问都通过 fp 来获取，因此编译器可以需要降低 sp，为运行时确定尺寸的变量分配空间。

注意，对于内部调用其他函数的函数，并且这个被调用函数使用太多的参数，以至必须使用堆栈传递参数，那么对堆栈帧内部的访问就需要 sp 的帮助了。

这样设计的一个很大好处在于，不管有堆栈帧指针的调用者函数，还是在内部被其调用的函数，都会对这部分进行特殊处理。对于被调用函数来说，因为 fp 寄存器是调用者函数的保全寄存器，因此必须保存 sp 寄存器的值，并在返回是恢复；这样，对于调用者函数，它所看到的堆栈帧中的这部分内容，不会出问题。

汇编工程师很高兴看到，编译器给函数的巨大参数结构保存空间后，还能使通过 alloca()分配空间而返回的地址处于 sp 之上。

有些工具还使用基于 fp 的堆栈帧，以便在本地变量很大而导致一些堆栈帧内容离 sp 太远时，能通过简单的 MIPS load/store 指令（只能访问±32KB 之间的内容）来访问这些内容。

现在再来看看前面一节例子的改进，主要添加了 alloca()的调用：

```
#include      <mips/asm.h>
#include      <mips/regdef.h>

#
#myfunc (arg1, arg2, arg3, arg4 ,arg5)
#
```

```
#framesize = locals + regsave(ra,s8,s0)  +  fregsave (f20/21) + args +pad
myfunc_frmsz = 4 + 12  +8 + (5*4) + 4

    .globl    myfunc
    .ent      myfunc , 0
    .frame    fp, myfunc_frmsz, $0

    subu      sp, myfunc_frmsz
    .mask     0xc0010000 , -4
    sw        ra , myfunc_frmsz-8 (sp)
    sw        fp , myfunc_frmsz-12  (sp)   #译者注：fp 就是 s8
    sw        s0 , myfunc_frmsz-16 (sp)
    .fmask    0x00300000 , -16
    s.d       $f20 , myfunc_frmsz – 24 (sp)
    move      fp , sp                      #save bottom of fixed frame
    -----
    # t6 = alloca ( t5 )
    addu      t5 , 7                       #make sure that size
    #and      t5 , ~7                      # is a multiple of 8
    #subu     sp , t5                      # allocate stack
    #addu     t6 , sp , 20                 #leave room for args
    -----
    < your code goes here , e.g.>
    # local = otherfunc(arg5,arg2,arg3,arg4,arg1)
    sw        a0 , 16 (sp)                      # arg5(out) = arg1(in)
    lw        a0 , myfunc_frmsz + 16 ( fp )     # arg1(out) = arg5(in)
    jal       otherfunc
    sw        v0 , myfunc_frmsz – 4   ( fp )    # local = result
    -----
    move      sp , fp                           #restore stack pointer
    l.d       $f20 , myfunc_frmsz – 24    (sp)
    lw        s0 , myfunc_frmsz – 16    (sp)
    lw        fp , myfunc_frmsz – 12    (sp)
    lw        ra , myfunc_frmsz – 8    (sp)
    addu      sp , myfunc_frmsz
    jr        ra
END (myfunc)
```

**看看修改过的地方：**

```
    .globl    myfunc
    .ent      myfunc , 0
    .frame    fp, myfunc_frmsz, $0
```

**这里不再使用** NESTED **宏，是因为使用了独立的堆栈帧指针，而这指针需要通过**.frame **直接明确地进行说明。后面需要修改** fp（**也就是** s8 **或**$30 **寄存器），因此必须在堆栈帧中保存：**

```
.mask    0xc0010000 , -4
sw       ra , myfunc_frmsz-8 (sp)
sw       fp , myfunc_frmsz-12  (sp)
sw       s0 , myfunc_frmsz-16 (sp)
```

接着通过 allocal()在堆栈中分配可变大小(t5B)的空间，并用寄存器 t6 指向这个空间：

```
# t6 = alloca ( t5 )
addu     t5 , 7                    #make sure that size
#and     t5 , ~7                   # is a multiple of 8
#subu    sp , t5                   # allocate stack
#addu    t6 , sp , 20              #leave room for args
```

注意，这里通过处理，将分配空间的尺寸调整到 8 的倍数，以便在满足分配空间要求的同时，使堆栈中内容的地址正确对齐。另外需要注意的是，在堆栈中留出了 20B 空间用于将来调用时存放参数。

在为其他函数建立参数时，使用 sp 寄存器，但是在访问自己的参数和本地变量时，需要使用 fp 寄存器：

```
sw       a0 , 16 (sp)                    # arg5(out) = arg1(in)
lw       a0 , myfunc_frmsz + 16 ( fp )   # arg1(out) = arg5(in)
jal      otherfunc
sw       v0 , myfunc_frmsz – 4  ( fp )   # local = result
```

最后，在函数返回前，恢复堆栈帧指针到在函数入口处所对应的寄存器，并恢复这个寄存器的内容（不要忘记恢复 fp 寄存器的旧值）：

```
move     sp , fp                         #restore stack pointer
l.d      $f20 , myfunc_frmsz – 24   (sp)
lw       s0 , myfunc_frmsz – 16   (sp)
lw       fp , myfunc_frmsz – 12   (sp)
```

## 10.10 可变长度参数列表

如果要建立一个参数数目不定的函数，需要用到相关工具的 stdarg.h 中定义的一套宏（ANSI 兼容工具必须具备的一套宏）。这套宏定义了 va_start(), va_end()和 va_arg()，具体使用，可以从 Algorithmics 的 SDE-MIPS 中实现的 printf()：

```
int   printf ( const char *format , ··· )
{
      va_start ( arg , format );
      n = vfprintf ( stdout , format , arg );
      va_end (arg );

      return n ;
}
```

一旦调用了 va_start()，就能解析出所有的参数。在给 printf()作参数格式转换的代码中，可以通过下面的方式获得下一个参数，这里假设这个参数是双精度浮点类型：

```
···
d = va_arg( ap , double );
···
```

千万不要在汇编程序中建立参数数目不定的函数，这会引起麻烦。

## 10.11 不同线程间共享函数和共享库的问题

C 库是包含了一些预编译的模块，在编译时，将被程序所用到的模块中函数和变量，动态链接到程序的二进制代码中。象 printf()一类标准的 C 函数，一般都会包含在 C 库中。

尽管 C 库提供了简单而强有力的方式扩展了 C 语言，但在多任务操作系统中，经常会引起麻烦。通常希望 C 库中的函数能和自己写的代码表现一样——就象是每个任务自己都拥有这部分代码拷贝。但是为了避免对同一内容多次拷贝而大量浪费内存空间，我们总是希望能够最低限度的共享 C 库中的函数代码。库函数很庞大：广泛使用的 X windows 系统的图形接口函数库就会给 MIPS 系统增加 300KB 的内容。

大多数 MIPS 操作系统提供不同任务间共享库代码的机制。为了解释共享函数的问题，下面介绍一下函数会用到的不同数据类型：

- 只读的数据和代码：所有能找到这些数据和代码的线程都可以共享
- 动态数据：特定线程能够安全地保持的参数、函数变量、保全寄存器和函数在堆栈中保持的其他信息。每个任务都要有自己的堆栈空间：即使是与其它线程共享一个地址空间的线程，也必须有自己的堆栈。在函数返回时，这些值都会消失。
- 静态暂时数据：在函数调用期间，不需要保留值的静态数据。原则上，如果不同的几个子函数共享一个静态暂时数据，可以用动态数据来取代它，只不过有点繁琐；这样意味着要重写代码，并重新进行编译。
- 线程范围数据：在库函数调用期间一直存在的静态数据，每个不同线程都要有这个数据的拷贝。全局 errno 变量就是这样的数据（errno 用来处理因调用 UNIX 风格文件 IO 操作函数而引起的出错代码）。
- 全局范围数据：库函数管理的用来跟踪系统状态变化的静态数据。一旦某个库函数为了保持多任务状态的信息而启用了这个数据，它就会成为操作系统的一部分，正常工作；这个内容已经超出了本书的范围。

这些数据类型的说明，在所有线程共用同一地址空间和每个线程使用单独地址空间的情况下，是有很大区别的。

### 10.11.1 单一地址空间的代码共享

在使用单一地址空间的操作系统，比如大多数实时系统，共享的库函数代码和数据是放在固定的地方；程序寻找库函数和库函数寻找自己的数据，都不会出现问题。然而，库必须可以重入：可供不同的任务使用，而且每个任务都可以在库的某个函数中悬挂，同时这个函数还可以被其他任务使用。动态数据是足够安全的，因此不需要拥有状态信息的简单任务不需要修改，也能工作正常。

静态暂时数据的访问可以通过信号量机制确保数据第一次访问到最后一次访问的连续性（详见 5.8.4）