

# 汇编器、链接器和 SPIM 仿真器

James R. Larus

微软研究院

对恶意中伤的恐惧，不能成为阻止言论和集会自由的借口。

——Louis Brandeis, *Whitney v. California*, 1927

## A.1 引言

将指令编码为二进制数字对计算机来说自然且有效。然而人类理解和处理这些数字有很大的困难。人们读和写符号（文字）比读和写一长串的数字容易多了。第 2 章说明了我们不需要在数字和文字之间做出选择，因为计算机指令可以具备很多种表达方式。人类可以读而且写这些符号，并且计算机可以执行等价的二进制数字。本附录描述了人类可读的程序被处理的过程：把一种程序形式转换成另外一种计算机可以执行的方式，提供了一些编写汇编程序的暗示，并且解释如何在 SPIM 上运行这些程序，SPIM 是一个执行 MIPS 程序的仿真器。SPIM 的 UNIX、Windows 以及 Mac OS X 版本在 CD 中可以得到。

汇编语言是计算机二进制编码——机器语言（machine language）的符号表示。汇编语言比机器语言更具备可读性，因为它使用符号而不是二进制数字。这些汇编语言中的符号名字通常以二进制模式出现，例如，操作码和寄存器指示符，所以可以阅读并记住它们。另外，汇编语言允许编程者使用 labels 来识别和指定保存指令或数据的内存字。

A-1  
A-3

### ○ 机器语言：在计算机系统内部通信的二进制表示。

一个被称为汇编器（assembler）的工具来将汇编语言转换成二进制指令。汇编器提供了比机器 0 和机器 1 更友好的表达，使得写程序和读程序都简化了。操作和地址的符号名称是这种表达方式的一个方面。另一个方面是编程设备增加了程序的清晰度。例如，A.2 节讨论的宏（macros）允许程序员通过定义新操作来扩展汇编语言。

- 汇编器：将符号版本的指令翻译成二进制版本的一段程序。
- 宏：一种模式匹配和替换机制，提供了简单的机制来命名经常使用的指令序列。

汇编器读入一个汇编语言的源文件，产生一个包含机器指令以及帮助将几个目标文件整合成一段程序的标签信息的目标文件。图 A-1-1 说明了如何构建一个程序。很多程序由多个文件组成——也被称为模块——这些文件被分开单独编写、单独编译、单独汇编。一个程序可能使用程序库中提供的预先写好的例程。一个模块通常包含到子例程的引用以及在别的模块和库中定义的数据。模块中的代码直到对其他目标文件或者库的标签的未确定的引用（unresolved reference）全部解决时才能执行。另一个工具被称为链接器（linker），将目标代码和库文件整合成一个可执行文件，这个文件是计算机可以执行的。

- 未确定的引用：一个需要从外部源代码获取更多信息才能完成的引用。

- ⑤ 链接器：也称为链接编辑器。是一个将独立的汇编机器语言程序组装起来，处理其中未定义的标签形成可执行文件的一个系统程序。

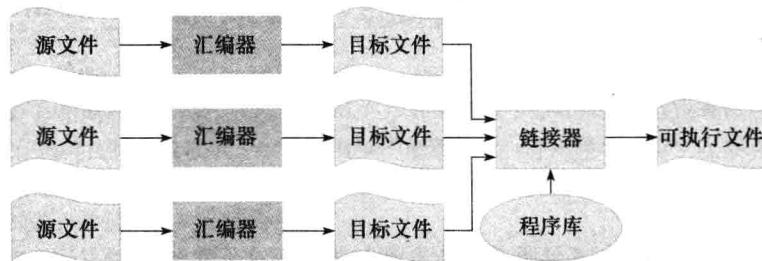


图 A-1-1 产生可执行文件的过程。一个汇编器将一个用汇编语言写的文件翻译成一个目标代码文件，这个目标代码文件又和其他的文件和库链接组成可执行文件

为了理解汇编语言的优势，考虑下面一系列图，这些图包含了一个短程序，这个程序计算而且打印出 0 ~ 100 的整数的平方和。图 A-1-2 展示一个 MIPS 计算机可以执行的机器语言。付出很多努力，你可以使用第 2 章指令表中的编码和指令格式来将指令转换成图 A-1-3 的类似的符号化程序。这个程序的形式是相当容易读的，因为操作和操作数使用符号，而不是使用二进制模式写的。然而，汇编语言仍然很难遵循，因为内存位置通过地址来指定而不是通过符号化标签来指定。

A-4

addiu	\$29, \$29, -32
sw	\$31, 20(\$29)
sw	\$4, 32(\$29)
sw	\$5, 36(\$29)
sw	\$0, 24(\$29)
sw	\$0, 28(\$29)
lw	\$14, 28(\$29)
lw	\$24, 24(\$29)
multu	\$14, \$14
addiu	\$8, \$14, 1
slti	\$1, \$8, 101
sw \$8,	28(\$29)
mflo	\$15
addu	\$25, \$24, \$15
bne	\$1, \$0, -9
sw	\$25, 24(\$29)
lui	\$4, 4096
lw	\$5, 24(\$29)
jal	1048812
addiu	\$4, \$4, 1072
lw	\$31, 20(\$29)
addiu	\$29, \$29, 32
jr	\$31
move	\$2, \$0

图 A-1-2 MIPS 用来计算和打印出 0 ~  
数的平方和的机器语言代码

同一个程序（图 A-1-2）的汇编语言版。然而，这个程序的代码没有标记寄存器或者内存地址，也没有包含注释。

图 A-1-4 展示了汇编语言使用记忆名称来标志内存地址指令。很多程序员喜欢以这种方式来读和写指令。那些名字前有个点，例如.data 以及.globl，是汇编指令（assembler directive），告诉汇编器如何翻译程序，但是不需要产生机器指令。名字后面跟一个冒号，如 str: 或者 main:，这些标签是下一个内存地址的名字。这个程序和汇编语言程序一样具有可读性（除了没有耀眼的注释），但是它还是很难遵循，因为需要很多简单的操作来完成简单的任务，因为汇编语言缺乏控制流结构，为程序的操作提供很少的暗示。

- 汇编指令：一个告诉汇编器如何翻译程序，但是不会产生机器指令的操作，通常它以圆点开始。

对比之下，图 A-1-5 的 C 程序不但短而且很清晰，因为具有记忆名字的变量和循环是显式的，而不是分支结构的。实际上，C 程序是唯一一个我们自己写的程序。其他形式的程序都是 C 编译器和汇编器产生的。

```
.text
.align 2
.globl main
main:
    subu    $sp, $sp, 32
    sw      $ra, 20($sp)
    sd      $a0, 32($sp)
    sw      $0, 24($sp)
    sw      $0, 28($sp)
loop:
    lw      $t6, 28($sp)
    mul   $t7, $t6, $t6
    lw      $t8, 24($sp)
    addu  $t9, $t8, $t7
    sw      $t9, 24($sp)
    addu  $t0, $t6, 1
    sw      $t0, 28($sp)
    ble   $t0, 100, loop
    la      $a0, str
    lw      $a1, 24($sp)
    jal   printf
    move  $v0, $0
    lw      $ra, 20($sp)
    addu  $sp, $sp, 32
    jr      $ra

.data
.align 0
str:
    .asciiz "The sum from 0 .. 100 is %d\n"
```

图 A-1-4 使用带有标签（label）的汇编语言写的同一个程序，但是没有注释。以圆点开始的指令是汇编指令（见 A.10 节）。.text 指示后续的行包含着指令。.data 指示出它们包含数据。.align n 指示后面这些行的元素应该是按照  $2^n$  来边界对齐的。因此，.align 2 就是下一个元素按照字对齐。.globl main 声明了 main 是一个全局的符号，应当对于其他文件中的代码来说是可见的。最后,.asciiz 保存了内存中的空终结符

```
#include <stdio.h>
int
main (int argc, char *argv[])
{
    int i;
    int sum = 0;
    for (i = 0; i <= 100; i = i + 1) sum = sum + i * i;
    printf ("The sum from 0 .. 100 is %d\n", sum);
}
```

图 A-1-5 使用 C 程序语言编写的程序

通常，汇编语言扮演两个角色（见图 A-1-6）。第一个角色是编译器的输出语言。编译器将使用高级语言（C 或者 Pascal）写的程序翻译成机器语言或者汇编语言表示的等价的程序。高级语言被称为源语言（source language），而编译器的输出是目标语言。

A-5

- 源语言：一种直接用来编写程序的高级语言。

汇编语言的另一个角色是作为一种编程语言。这个角色通常是它的主要功能。然而，今天由于大的内存以及更优良的编译器，很多程序员使用高级语言编写程序，而且很少看见计算机执行的指令。然而，当速度和面积很关键或者为了开发硬件特性，而高级语言中没有这些特性时，汇编语言仍然是很重要的。

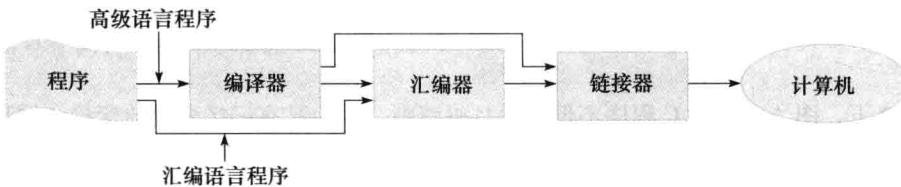


图 A-1-6 汇编语言由程序员编写或由编译器输出

虽然本附录关注 MIPS 汇编语言，但汇编语言编程在很多的其他机器上也是很相似的。CISC 机器（如 VAX）中的附加指令以及寻址模式，可使得汇编程序变短，但是不会改变程序的汇编流程，而且为汇编语言提供了高级语言的优势，例如，类型检测以及结构控制流。

### A. 1.1 什么时候使用汇编语言

与高级语言相比，使用汇编语言编程的主要原因是，它在速度和代码大小方面具有优势，而这两者极为重要。例如，一台计算机，它控制着机器的一个部分，如汽车刹车。一台计算机被合并到另一个设备中，例如一辆汽车，该计算机就被称作嵌入式计算机。这种类型的计算机需要对外部世界的事件做出快速的可预测的反应。由于编译器对操作所花费的时间引入了不确定性，程序员可能会发现很难保证高级语言编写的程序能在给定的时间间隔（传感器检测到轮胎打滑后的一毫秒内）做出响应。另一方面，一个汇编语言程序员，具有对指令执行的紧密控制。另外，在嵌入式应用中，减小了代码的体积，可使用更小的存储芯片，减小嵌入式计算机的代价。

一种混合的方法是，程序大部分用高级语言编写，时间关键部分用汇编语言编写，同时利用这两种语言。程序通常花费很多时间执行程序源代码中的很少一部分。这种发现就是 cache 中的局部性原理（见第 5 章的 5.1 节）。

程序分析测量一个程序在哪里花费了时间并找出其时间关键部分。大多数情况下，程序的这个部分可以使用更好的数据结构或者算法来实现。然而，有时候，显著的性能提高只能通过用汇编语言重写那段关键代码得到。

这种改进并不意味着高级语言的编译器就失效了。在为整个程序产生统一的高质量机器代码方面，编译器通常比程序员效果更好。然而，程序员比编译器在更深层次上理解程序的算法和行为，而且能够通过大量的努力和精巧的设计提高小段代码的质量。尤其，编程人员在编写代码时，同时考虑好几个子程序段。编译器通常单独编译一个程序段，而且必须遵循严格的规则，在程序段的边界处管理寄存器的使用。通过在寄存器中保存那些经常被使用的值，甚至跨越程序边界，编程人员可以使得程序运行得更快。

汇编语言的另一个主要优点是能够利用定制的指令——例如，字符串复制指令或者模式匹配指令。很多时候，编译器不能确定一个循环程序能不能被一条指令替代掉。然而，编写循环的那个程序员能够很容易地使用一个指令将其替换掉。

目前，由于编译技术的提高以及机器流水线的存在带来的复杂度（见第 4 章），程序员很难比编译器更具优势。

使用汇编语言的最后一个原因是，没有哪个高级语言能够适用于一个特定的计算机。很多老的或者定制的计算机没有编译器，所以编程人员唯一的选择就是汇编语言。

### A. 1.2 汇编语言的缺点

汇编语言的很多缺点极大地限制了它的广泛使用。也许它的主要缺点就是使用汇编语言

编写的程序本质上是针对特定机器的，而且如果需要在另一种计算机结构中运行，就必须重写。第1章讨论了计算机的快速发展，意味着体系结构变得过时。一个汇编语言程序仍旧和它的原始体系结构紧紧地绑定在一起，即使该计算机已被崭新、快速、性价比更高的机器所遮蔽。

汇编语言的另一个缺点是，汇编程序比等价的高级语言程序更长。例如，图A-1-5的C程序仅有11行，但是图A-1-4的汇编程序有31行。在更复杂的程序中，汇编和高级语言程序的比率（扩展因子）将更长，远不止像在这个例子中的3倍这样。不幸的是，实际的研究表明，程序员能够每天大约编写同高级语言行数一样多的汇编语言。这就意味着程序员使用高级语言大约会产生 $x$ 倍的生产率，这里的 $x$ 是汇编语言扩展因子。

长的程序更难阅读和理解，而且这些代码会包含更多的错误使得问题更为恶化。汇编语言使得这个问题恶化，因为它缺乏完备的结构。常见的编程用语，例如if-then语句和循环，汇编语言必须通过分支和跳转来实现。导致程序变得很难读懂，因为读者必须从汇编语言的每一句来为每个高级语言结构重建，这是很困难的。例如，看图A-1-4并回答下面的问题：使用的什么循环？它的下界和上界分别是什么？

**01 精解** 不需要汇编器，编译器直接产生机器语言。与使用汇编器作为编译的一部分的那些编译器相比，这些编译器通常执行得更快。然而，产生机器码的编译器必须执行一个汇编器通常执行的那些任务，例如，翻译地址，将指令编码成二进制数字。在编译速度和编译器的简洁性之间进行折中。

**01 精解** 尽管有这些考虑，一些嵌入式应用使用高级语言编写。很多这样的应用的程序很大，而且很复杂，这样的程序必须极其可靠。汇编语言程序相对于高级语言程序更长而且更难编写。这极大地增加了使用汇编语言编写程序的代价，使得验证这些程序的正确性极其困难。事实上，这些考虑导致为这些嵌入式系统埋单的国防部门开发Ada——一种编写嵌入式系统的新高级语言。

## A.2 汇编器

汇编器将汇编语言文件翻译成二进制机器指令和二进制数据组成的文件。翻译过程有两个主要步骤。第一步，找到标签(label)对应的内存地址，因此符号名字和地址之间的关系在指令被翻译的时候就确定了。第二步，将每个汇编语句的数字化的操作码、寄存器指示器和标签翻译成合法的指令。如图A-1-1所示，汇编器产生一个输出文件，叫作目标文件，目标文件包含机器指令、数据和书签信息。

目标文件通常不能被执行，因为它引用了其他文件中的过程或数据。如果标签目标可以被定义它的文件之外的其他文件所应用，那么这个就是外部标签(external label)（也称为全局标签）。如果仅仅能在定义它的文件内部被引用，则标签是局部的。很多的汇编器，默认标签是局部的，而且必须是显式声明为全局的。子程序和全局变量需要外部标签，因为它们在一个程序中被很多文件所引用。局部标签(local label)隐藏了对别的模块不可见的名字——例如，C中的静态函数仅仅被同一个文件中的函数所调用。另外，编译产生的名字——例如，一个循环的开始处的指令的名字——就是局部的，这样编译器就不需要为每个文件产生唯一的名字。

② 外部标签：也称为全局标签。标签对应一个目标，这个目标可以在定义这个标签的文件之外被引用。

A-9

A-10

◎ 局部标签：标签对应一个目标，这个目标仅仅可以被定义这个标签的文件内部引用。

### 01 例题·局部和全局标签

考虑图 A-1-4 中的程序。子程序具有一个外部标签（全局）main，它包含了两个局部标签——loop 和 str——它们仅仅在这个汇编文件中是可见的。最终，包含一个对外部标签 printf 来说未处理（unresolved）的引用，printf 是一个打印数值的库程序。图 A-1-4 中的标签能否从另一个文件引用？

### 01 答案

仅全局标签在外部是可见的，所以仅 main 标签可以在外部被引用。 □

因为汇编器独立地处理每一个文件，它仅知道每个局部标签的地址。汇编器依赖于别的工具，如利用链接器（linker）将目标文件以及库文件整合起来形成可执行文件，并且将外部标签处理掉。汇编器通过提供标签的列表以及未处理的引用来辅助链接器工作。

然而，局部标签对汇编器来说还是个令人感兴趣的挑战。和大部分高级语言中的名字不同，汇编标签可能在它们定义之前就使用。例如，在图 A-1-4 中，标签 str 在定义之前就被 la 指令使用。一个前向引用（forward reference）的可能性，就像前面这个例子，强迫汇编器将一个程序的翻译过程分成两步：首先找到所有的标签，然后产生指令。例如，当汇编器看到指令 la，它不知道这个标签为 str 的字在哪里，或者甚至不知道 str 这个标签到底是指令还是数据。

A-11

◎ 前向引用：一个标签在被定义之前就被使用。

第一遍，汇编器将汇编文件的每一行读入，将其分解成几个部分。这些部分叫作词汇单位，都是独立的字、数字和标点符号。例如，下面的行

ble \$t0, 100, loop

包含 6 个词汇单元：ble 指令的操作码、寄存器说明符 \$t0、逗号、数字 100、逗号，还有符号 loop。

如果一行以标签作为开始，汇编器在它的符号表（symbol table）中记录标签的名字，以及指令在内存中占据的内存字的地址。汇编器接着计算当前行中这个指令内存中占据多少个内存字。通过跟踪指令大小，汇编器可以确定下一条指令在哪里。为了计算一个可变长度的指令大小，例如在 VAX 中，汇编器必须仔细地确定这些。然而，对于固定长度的指令，像 MIPS 中的那些，仅仅需要粗略计算。汇编器采用类似的办法计算数据语句需要的空间。当汇编器到达一个汇编文件末尾时，符号表记录了文件中每个标签的位置。

◎ 符号表：用来将标签的名字和指令占用的内存字的地址相匹配的一个表。

第二遍，汇编器使用整个文件的符号表中的信息，在这一遍产生机器代码。汇编器再一次检查文件中每一行。如果一行中包含了指令，汇编器将其指令码和操作数（寄存器指示器或者内存地址）组合成一条合法的指令。这个过程和第 2 章的 2.5 节的做法很相似。引用在另一个文件中定义的外部标签指令和数据字不能完全地汇编（因为它们是未决的），因为符号的地址不在符号表中。汇编器确实不能对这些未决的引用发牢骚，因为对应的标签很可能在另一个文件中定义。

**01 重点** 汇编语言是一种编程语言。它和高级语言（如 BASIC、Java 和 C）的主要不同是

汇编语言提供了很少、简单的数据以及控制流。汇编语言程序不能指定一个变量中的数据类型。相反，编程人员必须对一个值使用恰当的操作（例如，整数或者浮点加法）。另外，在汇编语言中，程序的所有控制流必须使用 go to 实现。这两个因素使得汇编语言编程对于任何机器——MIPS 或者 x86——比使用高级语言编程更困难而且更容易出错。

A-12

- 01 精解** 如果汇编器的速度比较重要，两步的过程可以采用反向修补（backpatching）技术一次遍历汇编文件来实现。在这一次遍历中，汇编器构建每个指令的一个（可能不完整的）二进制表示。如果指令引用了一个还没有定义的标签，汇编器在表中记录下这个标签和指令。当标签被定义后，汇编器查询这个表，找到包含对标签的所有前向（forward）引用的所有指令。汇编器回卷并校正它们的二进制表示，然后将它们并入标签地址，反向修补技术能加速汇编的原因在于：汇编器对输入只读一次。然而，它需要汇编器将程序的整个二进制表示保持在内存中，这样指令才可以被反复修补。这个需求会限制被汇编的程序大小。这个过程被那种具有几种类型的跨度范围不同的分支的机器复杂化了。当汇编器第一次在分支指令中见到没有处理的标签时，它必须要么使用最大的分支，要么冒险返回去，并且重新调整很多指令，以便为大的分支指令腾出位置。
- ② 反向修补：一种将汇编语言翻译成机器指令的办法，其中汇编器在第一遍扫描程序时就构建一个（可能不完整的）每个指令的二进制表示，然后返回对前面没有定义的标签进行替换。

### A.2.1 目标文件的格式

汇编器产生目标文件。UNIX 上的目标文件包含 6 个不同的部分（见图 A-2-1）：

- 目标文件头描述了文件中其他段的大小和位置。
  - 代码段（text segment）包含了源文件中程序的机器语言代码。这些程序可能是不可执行的，因为包含了未处理的引用。
  - 数据段（data segment）包含了源文件中数据的二进制表示。数据可能是不完整的，因为未解决的引用可能包含在其他文件中。
  - 重定位信息（relocation information）指明指令和数据字依赖于绝对地址（absolute address）。如果程序的这些部分在内存中被移动，这些引用必须改变。
  - 符号表中包含源文件中外部标签对应的地址，列出未处理的引用。
  - 调试信息包含了被编译的程序的简洁描述，这样调试器可以找到源文件中对应行的指令地址，而且打印出可读形式的数据结构。
- ② 代码段：UNIX 目标文件的一个段，包含源文件中程序的机器语言代码。
  - ② 数据段：UNIX 目标文件或者可执行文件的一个段，包含程序初始所使用的数据的二进制表示。
  - ② 重定位信息：UNIX 目标文件的一个段，根据绝对地址来区别数据字和指令。
  - ② 绝对地址：内存中变量或者程序的实际地址。

目标文件头    代码段       数据段    重定位信息    符号表    调试信息

图 A-2-1 目标文件。UNIX 的汇编器产生一个具有 6 个不同段的目标文件

A-13 汇编器产生包含程序和数据的二进制表示的目标文件，以及其他有助于将程序的片段连接起来的信息。

重定位信息是必要的，因为当一个程序片段或者数据块和程序剩余的部分链接后，汇编器不知道这些程序或者代码将会被存放到内存的什么位置。一个文件的程序和数据被保存在内存中的一个连续的区域，但是汇编器不知道这段内存如何定位。汇编器还会将一些符号表入口传递给链接器。尤其，汇编器必须记录哪个外部符号在一个文件中定义，这个文件中哪些引用没有解决。

**01 精解** 为方便起见，汇编器假设每个文件以相同的地址开始（例如，地址 0），当它们在内存中分配地址时，期望链接器把代码和数据重新定位。汇编器产生重定位信息，这些信息包含一个入口，描述文件中的每个指令或数据字，这些字访问绝对地址。对于 MIPS，仅仅子程序调用、装载和保存指令引用绝对地址。例如分支，使用指令 PC 相对寻址，不需要定位。

## A. 2.2 附加工具

汇编器提供一类方便的特性帮助汇编程序变得短而且容易写，但是没有从根本上改变汇编语言。例如，数据布局指令允许一个编程者来描述以一种比二进制方式更简明和自然的方式来表示数据。

在图 A-1-4 中，指令

```
.asciiz "The sum from 0 .. 100 is %d\n"
```

在内存中保存字符串的字符。将这条指令和它的各个字符的 ASCII 值（这些字符的 ASCII 表示见第 2 章的图 2-15）进行比较：

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

A-14 .asciiz 描述更容易读懂，因为它使用字母表示字符，而不是使用二进制数字。汇编器能够比人更快速且更准确地将字符转换成它们的二进制表示。数据布局指令指定一个人类可读的数据格式，汇编器将其转换成二进制。其他字符串指令布局指令在 A. 10 节描述。

**01 例题・使用这个指令定义一串字节：**

```
.asciiz "The quick brown fox jumps over the lazy dog"
```

**01 答案**

```
.byte 84, 104, 101, 32, 113, 117, 105, 99
.byte 107, 32, 98, 114, 111, 119, 110, 32
.byte 102, 111, 120, 32, 106, 117, 109, 112
.byte 115, 32, 111, 118, 101, 114, 32, 116
.byte 104, 101, 32, 108, 97, 122, 121, 32
.byte 100, 111, 103, 0
```

□

宏（macro）是一种模式匹配和替换工具，提供一种简单的机制来命名一个经常使用的指令序列。不用每次使用同样的指令时重复输入，程序员只要启动宏，汇编器使用对应的指令序列替换这个宏调用。与子程序类似，宏允许程序员为一个公用操作产生和命名一个新的抽象。和子程序不同的是，宏不会导致一个子程序调用，也不会在程序运行时返回，因为一个宏的调

用会在程序汇编的时候被一个宏体替换。在替换完毕后，产生的汇编程序和没有使用宏的对等程序没有区别。

### 01 例题·宏

例如，假设程序员需要打印很多数字。一个库例程 `printf` 接受一个格式化的字符串，以及一个或更多个要打印的值作为其参数。程序员能够使用下面的指令打印出寄存器 \$7 中的整数：

```
.data
int_str: .asciiz "%d"
.text
la    $a0, int_str # Load string address
                  # into first arg
mov   $a1, $7   # Load value into
                  # second arg
jal   printf   # Call the printf routine
```

`.data` 指令告诉汇编器将字符串保存到程序的数据段，而且 `.text` 指令告诉汇编器将指令保存到代码段。

然而，以这种方式打印很多数字（程序写起来）相当乏味，而且产生的冗长的程序让人很难读懂。一种可供选择的办法是引入宏，`print_int`，来打印一个整数：

```
.data
int_str:.asciiz "%d"
.text
.macro print_int($arg)
la $a0, int_str # Load string address into
                  # first arg
mov $a1, $arg   # Load macro's parameter
                  # ($arg) into second arg
jal printf     # Call the printf routine
.end_macro
print_int($7)
```

宏有一个形式参数（formal parameter）`$arg`，它是用来为宏的参数命名。当宏被展开时，贯穿宏体的形式参数被来自调用的参数替换。之后汇编器使用最新扩展的宏体替换这个宏调用。对于第一次 `print_int` 的调用，参数是 `$7`，所以宏扩展成以下代码：

```
la $a0, int_str
mov $a1, $7
jal printf
```

② 形式参数：过程或者宏的参数变量，一旦这个变量被参数替换，宏就被展开。

在第二次调用 `print_int` 时，也就是说，`print_int($t0)`，参数是 `$t0`，宏被展开为：

```
la $a0, int_str
mov $a1, $t0
jal printf
```

调用 `print_int($a0)` 展开后的结果是什么？

### 01 答案

```
la $a0, int_str
mov $a1, $a0
jal printf
```

这个例子暴露了宏的一个缺点。程序员使用宏必须意识到 `print_int` 使用寄存器 `$a0`，所以不能正确地打印那个寄存器的值。□

A-15

A-16

**01 硬件/软件接口** 一些编译器也实现了伪指令 (pseudoinstruction)，这是汇编器提供的指令，但是在硬件上没有实现。第 2 章包含很多 MIPS 汇编器如何综合伪指令和寻址方式的例子，该寻址方式来自 spartan MIPS 硬件指令集。例如，第 2 章的 2.7 节描述了汇编器如何从其他两个指令 (slt 和 bne) 综合 blt 指令。通过扩展指令集，MIPS 汇编器使得汇编语言编程更容易，而没有使硬件变得更复杂。很多伪指令能够使用宏来模拟，但是有这些指令，MIPS 汇编器能产生更好的代码，因为它使用专用的寄存器 (\$at)，能够优化产生的代码。

**01 精解** 汇编器有条件地将代码汇编起来，这允许当汇编程序时，编程者可以将一组指令包含进去，或者将一组指令剔除出去。当几个版本的程序在一定程度上不同时，这个特性尤其有用。不是将这些程序放在单独的文件中——这样会将通用代码中的固定错误 (bug) 复杂化——编程者通常将几个版本融合成一个文件。代码的一个特定的版本被有条件地汇编，以使综合程序的其他版本时，这部分代码可排除在外。

如果宏和条件汇编有用，为什么 UNIX 系统很少提供汇编器？一个原因是，在这些系统上很多编程者使用像 C 这样的高级语言编写程序。大部分汇编代码由编译器产生，编译器发现重复代码比定义宏更方便。另一个原因是，UNIX 上的其他工具——例如，C 的预处理器 cpp，或者一个通用的宏处理器 m4——能提供汇编程序的宏定义以及条件汇编。

A-17

## A.3 链接器

**单独编译** (separate compilation) 允许程序被分割成多个片段，它们被保存在不同的文件中。每个文件包含一个逻辑相关的子程序以及数据结构组成的模块，这些文件形成一个大的程序。文件能够被编译，而且和其他的文件一样单独被汇编，所以一个模块的修改不需要重新编译整个程序。就像我们在上面讨论的，单独编译需要一个格外的链接步骤，以将单独的模块组成一个目标文件，将其未解决的引用解决。

单独编译：将程序划分成多个文件，每个文件被编译时，并不知道其他文件的信息。

将多个文件融合在一起的工具叫作链接器 (linker)（见图 A-3-1）。它执行三个任务：

- 为了寻找程序所使用的库程序而查询程序库。
- 为每个模块中的代码将要占用的内存确定内存地址，通过调整绝对引用，将这些指令重定位。
- 解决文件间的引用。

链接器的第一个任务是确保程序不包含没有定义的标签。链接器匹配外部的符号以及程序文件中未解决的引用。如果一个文件中外部符号和另一文件中的引用具有相同名字的标签，则未决的引用被确定。不匹配的引用意味着一个符号被使用，但是在程序的任何地方都没有定义。

在链接期间发现未解决的引用并不一定意味着程序员犯了错误。程序可能引用了一个库函数，该库函数的代码不在传递到链接器的目标代码中。在程序中匹配符号完毕后，链接器搜寻系统的程序库，目的是找程序中引用的预定义的子程序以及数据结构。基本库包含了读和写数据，分配和收回内存，执行数字操作。别的库包含访问数据库或者操作终端窗口。一个引用未解决符号的程序不在任何一个库中是错误的而且不能被链接。当程序使用了库例程，链接器从库中提取例程代码，并将其合并到程序的代码段。这个新的例程反过来可能依靠别的库例程，所以链接器继续读取别的库例程，直到没有外部引用是没有解决的或者没有哪个程序是不能被找到的。

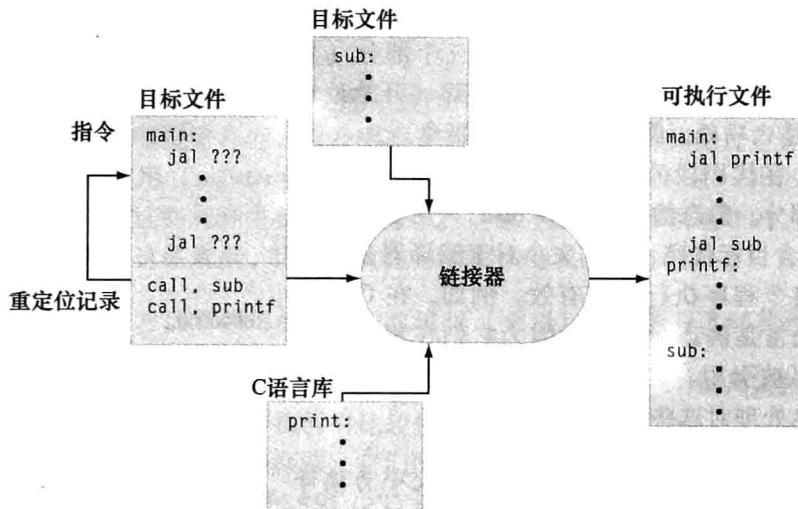


图 A-3-1 链接器搜查一组目标文件和程序库，寻找在程序中使用的非局部函数，将其合并成一个可执行文件，而且解决不同文件间的程序的引用

如果所有的外部引用被解决了，链接器接下来确定每个模块将占用的内存地址。因为文件在汇编上是独立的，汇编器不知道一个模块的指令或者数据相对于别的模块放在哪里。当链接器在内存中放一个模块时，所有的绝对引用必须重定位以便反映其真实的地址。因为链接器具有重定位信息来指出所有重定位引用，它能够高效地找到以及反向修补这些引用。

链接器产生一个可执行的文件，这个文件可以在计算机上运行。典型的，除了不包含没有被解决的引用或者重定位信息，这个文件具有和目标文件一样的格式。

A-18

## A. 4 加载

程序在链接阶段没有错误就可以运行。在运行之前，程序保存在像磁盘这样的二级存储的一个文件中。在 UNIX 系统中，操作系统核心将一个程序加载到内存并且开始运行。为了启动一个程序，操作系统执行以下步骤：

A-19

- 1) 读取可执行文件的头，目的是确定代码段和数据段的大小。
- 2) 产生程序的一个新地址空间。这个地址空间足够大，装得下代码段和数据段，还有堆栈段（见 A. 5 节）。
- 3) 将可执行文件中的指令和数据复制到一个新的地址空间。
- 4) 将传递给程序的参数复制到堆栈上。
- 5) 初始化机器寄存器。通常，大部分寄存器被清零，但是堆栈寄存器指针必须被赋值为堆栈地址的初始地址（见 A. 5 节）。
- 6) 跳转到一个启动程序，这个程序从堆栈中把程序的参数复制到寄存器，而且调用程序的 main 程序。如果 main 程序返回，启动程序退出系统调用，终止程序的执行。

## A. 5 内存的使用

下面几节描述本书前面提到的 MIPS 体系结构。前面几章主要关注硬件，以及硬件和低级软件的关系。这些章节主要关注汇编语言编程者如何使用 MIPS 硬件。描述在很多 MIPS 系统上的一组规则。很多情况下，硬件不会影响这些规则。相反，为了使不同的人编写的程序集合在

一起时能够工作，能有效地利用 MIPS 的硬件，这些规则代表了编程人员必须遵守的一种约定。

基于 MIPS 的系统通常将内存分割成三个部分（见图 A-5-1）。第一部分，接近地址空间的底部（开始地址是  $400000_{16}$ ），是代码段，保存的是程序的指令。

A-20 第二部分，在代码段的上面，称为数据段，它被进一步分割成两部分。静态数据（static data）（开始地址是  $10000000_{16}$ ）包含目标代码，它的大小对于编译器是已知的，其内容在整个程序执行期间有效。例如，在 C 语言中，全局变量通常是静态分配的，因为它们在程序执行的任何时候都可被引用。链接器既为静态的对象在数据段分配地址，也处理对这些对象的引用。

- ② 静态数据：包含数据的那部分内存，其大小为编译器所知，生命周期为整个程序的运行时间。

紧靠着静态数据之上的就是动态数据。这个数据，正如其名字所暗示的一样，是在程序执行过程中分配的。在 C 程序中，`malloc` 库例程发现并返回一个新的内存块。因为编译器不能预测一个程序将需要分配多大的内存，操作系统扩展了动态内存数据区域来满足这个需求。如图 A-5-1 中向上的箭头所指示的，`malloc` 通过使用系统调用 `sbrk` 扩展了动态区域，调用这个函数会导致操作系统在动态数据段之上为程序的虚拟地址空间加载更多的页（见第 5 章的 5.7 节）。

第三部分，程序堆栈段（stack segment）存在于虚拟地址空间的顶部（从地址  $7fffffff_{16}$  开始）。和动态数据相似，一个程序的堆栈段的最大尺寸不能够被预先知道。当程序向堆栈段压入变量时，操作系统会自动向下（数据段方向）扩展堆栈段。

- ② 堆栈段：程序用来保存过程调用帧的那段内存。

这种三段分割的内存格局不是唯一的格局。然而，它具备两个重要的特性：动态可以扩展的段尽量隔得很远，而且能够扩展，以便将整个程序的地址空间全部用完。

**01 硬件/软件接口** 由于数据段的起始地址远远高于程序的起始地址  $10000000_{16}$ ，存取指令不能直接使用它们的 16 位偏移域引用数据对象（见第 2 章的 2.5 节）。例如，为了加载位于数据段地址  $10010020_{16}$  的字到寄存器  $\$v0$  需要两条指令：

```
lui $s0, 0x1001 # 0x1001 means 1001 base 16
lw $v0, 0x0020($s0) # 0x10010000 + 0x0020 = 0x10010020
```

（数字之前的  $0x$  表示这个数字是十六进制的值。例如， $0x8000$  是  $8000_{16}$  或者  $32768_{10}$ 。）

为了在每个存取指令中避免重复 `lui` 指令，MIPS 系统通常使用一个专用的寄存器（ $\$gp$ ）作为全局指针指向静态数据段。这个寄存器包含了地址  $10008000_{16}$ ，所以存取指令可以使用 16 位的偏移来访问静态数据段的第一个 64KB。拥有这个全局指针，我们可以将以上例子改写为一条指令：

```
lw $v0, 0x8020($gp)
```

当然，一个全局指针寄存器使得寻址  $10000000_{16} \sim 10010000_{16}$  比别的堆地址定位要快。MIPS 编译器通常将全局变量存储在这个范围，因为这些变量具备固定的地址，而且比别的全局数据（例如数组）更合适。

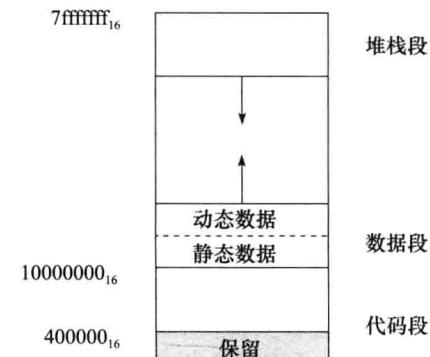


图 A-5-1 内存布局

## A. 6 过程调用规范

当程序中的过程被分别编译时，管理寄存器使用的规范是必要的。为了编译一个特定过程，编译器必须知道需要哪些寄存器，以及哪些寄存器的信息需要为其他过程保留。寄存器使用的规则被称作寄存器使用（register use）或者过程调用规范（procedure call convention）。顾名思义，大多数情况下，这些规则主要用于约束软件，而不是硬件必须遵守的。然而，很多编译器以及编程者努力遵循这些规范，因为违反这些规则会导致诡异的错误。

### ① 寄存器使用：或者称为过程调用规范。管理过程（调用）使用寄存器的软件协议。

本节描述的调用规范是 gcc 编译器所遵循的一个规范。MIPS 的原始编译器使用一个更为复杂的规范，而且这个规范会导致程序执行得比较快。

MIPS CPU 包含 32 个通用目的寄存器，它们的编号是 0 ~ 31。寄存器 \$0 的值总是 0。

- 寄存器 \$at (1)、\$k0 (26) 和 \$k1 (27) 是预留给汇编器和操作系统的，不能被用户程序或者编译器使用。
- 寄存器 \$a0 ~ \$a3 (4 ~ 7) 被用来传递初始 4 个参数到例程（其他的参数传递到堆栈中）。寄存器 \$v0 以及 \$v1 (2, 3) 被用来返回来自函数的值。
- 寄存器 \$t0 ~ \$t9 (8 ~ 15, 24, 25) 是调用者保存的寄存器（caller-saved register），被用来保存临时变量，这些值在调用的时候不需要保存（见第 2 章 2.8 节）。

A-21  
A-22

### ② 调用者保存的寄存器：调用程序保存的寄存器。

- 寄存器 \$s0 ~ \$s7 (16 ~ 23) 被称为被调用者保存的寄存器（callee-saved register），保存长期存活值，这些值应当在程序调用时保存。

### ③ 被调用者保存的寄存器：被调用者程序保存的寄存器。

- 寄存器 \$gp (28) 是一个全局指针，指向 64K 的静态数据内存块。
- 寄存器 \$sp (29) 是堆栈指针，指向堆栈的最后地址。寄存器 \$fp (30) 是数据帧指针。  
jal 指令写寄存器 \$ra (31)，是过程调用的返回地址。这两个寄存器将在下一节说明。

两个字母的缩写以及这些寄存器的名字——如 \$sp 代表的是栈指针——反映了寄存器在过程调用规范中所起的作用。在描述这样一个规范时，我们将使用寄存器的名字，而不是寄存器的编号。图 A-6-1 罗列了这些寄存器及其用途。

### A. 6.1 过程调用

本节描述一个程序（调用者，the caller）调用另一段程序（被调用者，the callee）的步骤。程序员使用像 C 或者 Pascal 这样的高级语言编程，从来都看不到一个程序调用另一个程序的细节。因为编译器负责低级的书签工作。然而，汇编语言程序员必须明确地实现每个程序调用和返回。

很多与调用相关的书签操作围绕着一个内存块，这个内存块被称为过程调用帧（procedure call frame）。这段内存被用作以下目的：

- ④ 过程调用帧：用来保存被调用过程的参数，保存可能会被过程修改的寄存器的值，但是这些寄存器的值不会被调用者所修改，并为被调用程序的局部变量提供空间。
  - 保持作为参数传递给过程的数值。
  - 保存一个过程可能会修改的寄存器，但是过程的调用者却不希望这些寄存器的值被修改。

寄存器名称	编号	使用规则	寄存器名称	编号	使用规则
\$zero	0	恒为 0	\$s0	16	保存临时值（过程调用预留）
\$at	1	为汇编器保留	\$s1	17	保存临时值（过程调用预留）
\$v0	2	表达式求值以及函数的结果	\$s2	18	保存临时值（过程调用预留）
\$v1	3	表达式求值以及函数的结果	\$s3	19	保存临时值（过程调用预留）
\$a0	4	参数 1	\$s4	20	保存临时值（过程调用预留）
\$a1	5	参数 2	\$s5	21	保存临时值（过程调用预留）
\$a2	6	参数 3	\$s6	22	保存临时值（过程调用预留）
\$a3	7	参数 4	\$s7	23	保存临时值（过程调用预留）
\$t0	8	临时（不为过程调用预留）	\$t8	24	临时（不为过程调用预留）
\$t1	9	临时（不为过程调用预留）	\$t9	25	临时（不为过程调用预留）
\$t2	10	临时（不为过程调用预留）	\$k0	26	为 OS 内核保留
\$t3	11	临时（不为过程调用预留）	\$k1	27	为 OS 内核保留
\$t4	12	临时（不为过程调用预留）	\$gp	28	全局区域的指针
\$t5	13	临时（不为过程调用预留）	\$sp	29	堆栈指针
\$t6	14	临时（不为过程调用预留）	\$fp	30	帧指针
\$t7	15	临时（不为过程调用预留）	\$ra	31	返回地址（函数调用使用）

图 A-6-1 MIPS 寄存器和使用规则

- 为过程的局部变量提供空间。

在大部分编程语言中，过程调用和返回遵循一个严格的后进先出（LIFO）的顺序，所以在一个栈中内存能被分配以及被再次分配，这就是为什么这些内存块有时称作堆栈帧。

图 A-6-2 展示了一个典型的堆栈帧。这个帧由以下部分组成：帧指针（\$fp），这个指针指向这个帧的第一个字；堆栈指针（\$sp），指向帧的最后一个字。栈从内存的高地址开始向下增长，所以帧的指针指向堆栈指针的上面。一个过程的执行使用帧指针来快速地访问堆栈帧中的值。例如，一个堆栈帧中的参数可以使用以下命令来加载到寄存器 \$v0：

`lw $v0, 0($fp)`

堆栈帧可以有好几种不同的构建方式；然而，调用者和被调用者必须遵从一系列步骤。下面来描述在大部分 MIPS 机器上使用的调用规范步骤。这个规范在过程调用中的三个阶段出现：在调用者激活被调用者之前，被调用者开始执行，并且被调用者返回调用者之前。在第一种情况下，调用者将过程调用参数放在指定的地方，激活被调用者做如下事情：

1) 传递参数。根据规范，第一批的 4 个参数被传递到寄存器 \$a0 ~ \$a3。任何剩余的参数将被压入堆栈中，而且出现在被调用过程的堆栈帧的开始。

2) 保存调用者寄存器。被调用过程可以直接使用这些寄存器（\$a0 ~ \$a3 以及 \$t0 ~ \$t9），而不需要首先保存这些寄存器的值。如果调用者在调用之后还想使用这些寄存器，那么它必须在调用之

A-23

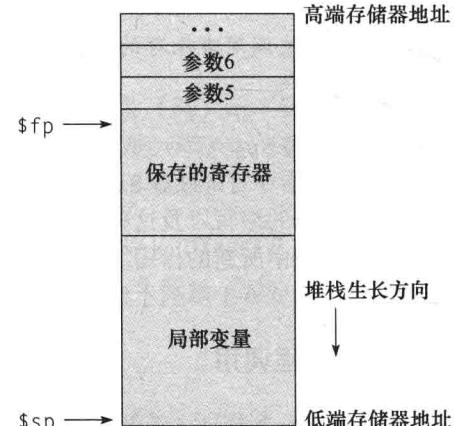


图 A-6-2 堆栈帧的示意图。帧指针（\$fp）指向当前执行过程的堆栈帧的第一个字。堆栈指针（\$sp）指向该帧的最后一个字。最前面 4 个参数被传递到寄存器中，所以第五个参数成为栈中第一个被保存的参数

前保存寄存器的值。

3) 执行一个 `jal` 指令（见第 2 章的 2.8 节），这个指令跳转到被调用者的第一个指令并将返回地址保存到寄存器 `$ra` 中。

A-24  
A-25

在一个被调用的例程开始运行之前，必须经过以下几步来建立它的堆栈帧：

- 1) 为帧分配内存空间，通过栈指针来减小帧的大小。
- 2) 在帧中保存被调用者的寄存器。调用者必须在修改这些寄存器之前保存这些寄存器的值（`$s0 ~ $s7`、`$fp` 和 `$ra`），由于调用者期望发现这些寄存器在调用之后保持不变。寄存器 `$fp` 被每个过程保存，这个指针为过程分配一个新的堆栈帧。然而如果被调用者调用别的程序，寄存器 `$ra` 仅仅需要被调用者保存。别的被调用者保存的寄存器被使用的话，也必须被保存。

3) 设置栈帧指针，其值为栈帧大小减去 4 加上 `$sp`，保存在寄存器 `$fp` 中。

**01 硬件/软件接口** MIPS 寄存器使用规范提供被调用者以及调用者保存的寄存器，因为这两种类型的寄存器在不同的环境中各具优势。被调用者保存的寄存器最好用来保存生存期长的值，例如来自用户程序的变量。如果被调用者期望使用这个寄存器，这个寄存器就仅仅在过程调用中被保存。另一方面，调用者保存的寄存器最好被用来保存短期存在的量，这些值在过程调用中不长期存在，例如地址计算中的立即值。在一个过程调用中，被调用者也可以使用那些保存临时值的寄存器。

最终，通过执行以下几步，被调用者返回到调用者：

- 1) 被调用者是一个具备返回值的函数，就将返回值放到寄存器 `$v0`。
- 2) 恢复所有被调用者保存的寄存器，这些寄存器保存前一个过程的入口。
- 3) 向 `$sp` 加上帧大小，将帧从栈中弹出。
- 4) 跳转到寄存器 `$ra` 中的地址处。

**01 精解** 编程语言不允许递归过程（recursive procedure）——一个过程通过一串调用，可以间接或者直接地调用自己——不需要在堆栈中分配帧。在一个非递归的语言中，每个过程的帧可能被静态分配，因为在同一时间，仅仅允许一个过程处于活动状态。旧版本的 Fortran 禁止递归，在一些比较老的机器中静态分配帧产生代码比较快。然而，在类似 MIPS 这样的存取体系结构中，堆栈帧的速度也可能很快。因为一个堆栈指针寄存器直接指向活动堆栈帧，这允许一个存取指令访问这个帧中的值。另外，递归是一种很有价值的编程技巧。

② 递归过程：就是指某个过程能通过调用链直接或间接地调用自己。

A-26

## A.6.2 过程调用举例

作为一个例子，考虑以下 C 程序：

```
main ()
{
    printf ("The factorial of 10 is %d\n", fact (10));
}

int fact (int n)
{
    if (n < 1)
        return (1);
    else
        return (n * fact (n - 1));
}
```

这个函数计算而且打印  $10!$  ( $10!$  的阶乘,  $10! = 10 \times 9 \times \dots \times 1$ )。fact 是一个递归例程, 计算  $n!$ , 通过对  $n$  乘以  $(n-1)!$ 。这段代码对应的汇编代码说明程序如何管理堆栈帧。

在入口上, 例程 main 创建一个堆栈帧, 而且保存被调用者将会修改的两个寄存器: \$fp 和 \$ra。一个帧的大小比两个寄存器大, 因为调用规范所需的一个堆栈帧的最小大小是 24 字节。最小的帧可以容纳 4 个寄存器参数 (\$a0 ~ \$a3) 以及 \$ra 的返回地址, 再加上一个双字边界 (一共 24 字节)。由于 main 函数也需要保存 \$fp, 它的堆栈帧必须是两个字大小 (记住: 堆栈指针保持双字对齐)。

```
.text
.globl main
main:
    subu    $sp,$sp,32      # Stack frame is 32 bytes long
    sw      $ra,20($sp)     # Save return address
    sw      $fp,16($sp)     # Save old frame pointer
    addiu   $fp,$sp,28      # Set up frame pointer
```

main 程序然后调用阶乘例程而且将它的唯一参数 10 传给阶乘函数。在 fact 函数返回后,

A-27] main 调用 printf, 而且给 printf 传递一个格式化字符串, 以及从 fact 返回的结果两个参数。

```
li      $a0,10          # Put argument (10) in $a0
jal    fact             # Call factorial function

la      $a0,$LC          # Put format string in $a0
move   $a1,$v0            # Move fact result to $a1
jal    printf           # Call the print function
```

最终, 在打印出阶乘结果后, main 返回。但是首先, 它必须恢复以下这些寄存器的值, 将它们从栈中弹出:

```
lw      $ra,20($sp)     # Restore return address
lw      $fp,16($sp)     # Restore frame pointer
addiu  $sp,$sp,32      # Pop stack frame
jr      $ra              # Return to caller

.rdata
$LC:
.ascii  "The factorial of 10 is %d\n\000"
```

阶乘函数的结构和 main 函数很相似。首先, 阶乘函数创建一个堆栈帧, 把它可能会使用的被调用者寄存器保存起来。另外, 还保存 \$ra 以及 \$fp, fact 函数也保存它的参数 (\$a0), 这个参数在递归调用的时候会被使用:

```
.text
fact:
    subu   $sp,$sp,32      # Stack frame is 32 bytes long
    sw      $ra,20($sp)     # Save return address
    sw      $fp,16($sp)     # Save frame pointer
    addiu  $fp,$sp,28      # Set up frame pointer
    sw      $a0,0($fp)      # Save argument (n)
```

fact 例程的核心执行 C 程序计算。这个函数测试它的参数是否比 0 大。如果不是, 例程返回值 1。如果参数比 0 大, 例程递归地调用它自己, 计算  $\text{fact}(n-1)$  而且乘以  $n$ :

```
lw      $v0,0($fp)      # Load n
bgtz  $v0,$L2            # Branch if n > 0
li      $v0,1              # Return 1
jr      $L1                # Jump to code to return

$L2:
    lw      $v1,0($fp)      # Load n
    subu   $v0,$v1,1        # Compute n - 1
    move   $a0,$v0            # Move value to $a0
```

A-28]

```

jal      fact          # Call factorial function

lw       $v1,0($fp)    # Load n
mul     $v0,$v0,$v1    # Compute fact(n-1) * n

```

最后，阶乘函数恢复被调用者保存的那些寄存器而且返回寄存器 \$v0 中的值：

```

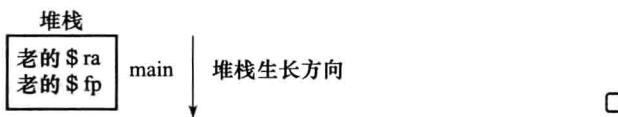
$L1:                      # Result is in $v0
lw      $ra, 20($sp)  # Restore $ra
lw      $fp, 16($sp)  # Restore $fp
addiu $sp, $sp, 32  # Pop stack
jr      $ra             # Return to caller

```

### 01 例题·递归过程中的栈

图 A-6-3 展示了 fact(7) 的调用栈。main 最先运行，所以它的帧在栈的最深处。main 调用了 fact(10)，它们的栈帧挨着。每个调用，递归调用 fact 函数来计算更低一级的阶乘。栈帧和这些函数的调用按照 LIFO 的顺序并行着。当 fact(10) 返回的时候，栈看起来是什么样子？

### 01 答案



**01 精解** MIPS 编译器和 gcc 编译器之间的差异是，MIPS 编译器通常不需要帧指针，所以这个寄存器可作为另一个被调用者保存寄存器 \$s8 使用。这种改变在过程调用和返回序列中节省了一对指令。然而，这使得代码产生变得复杂，因为一个过程必须使用 \$sp 来访问栈帧，如果有数值被压到栈中，它的值可以在一个过程执行中变化。



图 A-6-3 调用 fact(7) 过程中的栈帧

## A.6.3 另外一个过程调用的例子

作为另一个例子，考虑下面的程序，它计算 tak 函数，这是一个被广泛使用的基准测试程序，由 Ikuo Takeuchi 创建。这个函数不计算任何有用的东西，但这个函数是深度的递归程序，用它可以说明 MIPS 调用的规范。

```

int tak (int x, int y, int z)
{
    if (y < x)
        return 1+ tak (tak (x - 1, y, z),
                     tak (y - 1, z, x),
                     tak (z - 1, x, y));
    else
        return z;
}

int main ()
{
    tak(18, 12, 6);
}

```

这段程序的汇编代码将在下面展示。tak 函数首先保存它的返回地址到堆栈帧中，而且将

A-30 它的参数保存在被调用保存的寄存器中，因为例程或许会调用那些需要使用寄存器 \$a0 ~ \$a2 以及 \$ra 的例程。函数使用被调用者保存的寄存器，由于它们在函数的整个生命期中保持有效，这期间包含几个可能会修改寄存器值的函数调用。

```
.text
.globl tak

tak:
    subu    $sp, $sp, 40
    sw      $ra, 32($sp)

    sw      $s0, 16($sp)    # x
    move   $s0, $a0
    sw      $s1, 20($sp)    # y
    move   $s1, $a1
    sw      $s2, 24($sp)    # z
    move   $s2, $a2
    sw      $s3, 28($sp)    # temporary
```

通过测试，如果  $y < x$ ，例程开始执行。否则，分支转到标签 L1 处，如下所示。

```
bge    $s1, $s0, L1    # if (y < x)
```

如果  $y < x$ ，那它就执行例程的主体，主体包含了 4 个递归的调用。第一个调用使用几乎和它的母体相同的参数：

```
addiu $a0, $s0, -1
move  $a1, $s1
move  $a2, $s2
jal   tak          # tak (x - 1, y, z)
move  $s3, $v0
```

注意到，第一个递归调用的结果被保存到寄存器 \$s3，这样便于不久后使用。

函数现在为第二个递归调用做准备。

```
addiu $a0, $s1, -1
move  $a1, $s2
move  $a2, $s0
jal   tak          # tak (y - 1, z, x)
```

在下面的指令中，来自递归调用的结果被保存到寄存器 \$s0。但是首先，也是最后一次，

A-31 我们需要读这个寄存器的值，其中保存第一个参数的值。

```
addiu $a0, $s2, -1
move  $a1, $s0
move  $a2, $s1
move  $s0, $v0
jal   tak          # tak (z - 1, x, y)
```

在三个内部递归调用之后，我们准备最后的递归调用。调用之后，函数的结果保存到 \$v0 中，控制函数流程的跳转。

```
move  $a0, $s3
move  $a1, $s0
move  $a2, $v0
jal   tak          # tak (tak(...), tak(...), tak(...))
addiu $v0, $v0, 1
j     L2
```

标签 L1 处的代码是一个 if-then-else 语句序列。它仅仅将参数 z 的值传递到返回寄存器而落入到函数格局中。

```
L1:
    move  $v0, $s2
```

以下的代码是函数末尾，它恢复被保存的寄存器并将函数值返回给它的调用者。

```
L2:
lw      $ra, 32($sp)
lw      $s0, 16($sp)
lw      $s1, 20($sp)
lw      $s2, 24($sp)
lw      $s3, 28($sp)
addiu   $sp, $sp, 40
jr      $ra
```

main 函数使用最初的参数来调用 tak 函数，然后得到计算结果 result (7) 而且使用 SPIM 系统调用来打印整数的值。

```
.globl  main
main:
    subu   $sp, $sp, 24
    sw     $ra, 16($sp)

    li     $a0, 18
    li     $a1, 12
    li     $a2, 6
    jal    tak          # tak(18, 12, 6)

    move   $a0, $v0
    li     $v0, 1           # print_int syscall
    syscall

    lw     $ra, 16($sp)
    addiu  $sp, $sp, 24
    jr      $ra
```

A-32

## A.7 异常和中断

第 4 章的 4.9 节描述了 MIPS 异常机制，包括指令执行中发出错误导致的异常以及 I/O 设备引起的外部中断。本节描述异常以及中断处理（interrupt handling）的更多细节。<sup>②</sup>在 MIPS 处理器中，CPU 中一个被称为 coprocessor 0 的部分记录软件处理异常和中断所需要的信息。MIPS 仿真器 SPIM 没有实现 coprocessor 0 的寄存器，因为在一个仿真器中，不需要很多寄存器，或者寄存器不是内存系统的一个部分，SPIM 就没有实现它。然而，SPIM 确实提供了下列 coprocessor 0 的寄存器：

② 中断处理：一段代码，作为异常或者中断的执行结果。

A-33

寄存器名称	寄存器编号	使用
BadVAddr	8	一个会发生内存引用冲突的内存地址
Count	9	计时器
Compare	11	一个用来和计时器进行比较，当它的值和计时器匹配时，会发生中断
Status	12	中断掩码以及使能位
Cause	13	异常类型以及中断挂起位
EPC	14	引起异常的指令地址
Config	16	机器的配置

② 本节讨论 MIPS-32 体系结构中的异常，这些异常是 SPIM 的 7.0 及以后的版本所实现的。SPIM 的早期版本实现了 MIPS-1 体系结构，但是其中对异常的处理有些不同。将这些不同版本的程序转变成在 MIPS-32 上运行不是一件困难的事情，因为只有状态和原因寄存器的域需要改变，使用 rfe 指令替换 eret 指令。

这 7 个寄存器是 coprocessor 0 处理器的寄存器组的一部分。它们通过 mfc0 以及 mtco 指令来访问。异常之后，寄存器 EPC 包含了在执行时发生异常的那条指令的地址。如果异常是外部中断引起的，那么指令将不需要重新开始执行。除了导致问题的指令处于分支或跳转指令的延迟槽中之外，所有其他的异常均由执行 EPC 处的指令引起。在那种情况下，EPC 指向分支或者跳转指令而且原因寄存器中的 BD 位被设置。当这些位设置好后，异常处理函数必须查看引起异常的 EPC +4。然而，在别的情况下，异常处理函数通过返回到指令的 EPC 地址处恢复被中断的程序。

如果指令所引起的异常导致一个内存访问，寄存器 BadVAddr 包含被引用的内存地址的地址。

Count 寄存器是一个计数器，当 SPI 模块运行时，它按照一定的频率递增（默认，每 10 毫秒一次）。当 Count 寄存器中的值和比较寄存器中的值匹配时，处于 5 级的硬件中断就会发生。

图 A-7-1 展示了 MIPS 仿真器 SPIM 实现的状态寄存器域。interrupt mask 域为 6 个硬件包含了 6 个位和 2 个软件中断层次。如果掩码位的值是 1，就是允许处理器这个级别上的中断；如果掩码位的值是 0，就是不允许处理器这个级别上的中断。当中断到达时，中断在其原因寄存器中设置中断挂起位，即使掩码位是无效的。当一个中断被挂起时，当随后其掩码位被允许时，它会中断处理器。

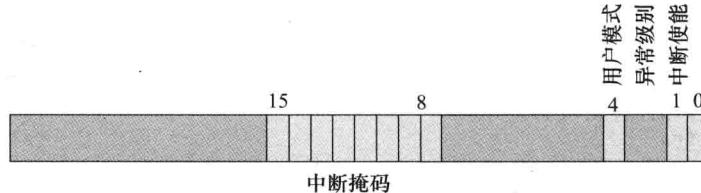


图 A-7-1 状态寄存器

当处理器运行在核心模式，用户模式位是 0；如果用户模式位是 1，则说明处理器处于用户态。对于 SPIM，这一位固定是 1，因为 SPIM 处理器没有实现核心模式。异常级别位通常是 0，但是当异常发生时就被设置成 1。当这一位是 1 时，中断被禁止，而且如果另一个异常发生，EPC 也不会被更新。这一位阻止一个异常处理被别的中断或者异常打断，但是它应当在异常处理结束时可以复位。如果 interrupt enable 位是 1，中断就被允许。如果这一位是 0，它们就被禁止。

图 A-7-2 展示了 SPIM 中的原因寄存器字段的子集。如果最后一个异常发生时，一个正在执行的指令处于分支延迟槽中，分支延迟位则为 1。当一个中断处于给定的硬件或者软件层时，中断挂起位变成 1。异常代码寄存器通过以下代码描述了一个异常的原因：

编号	名称	异常产生的原因
0	Int	中断（硬件）
4	AdEL	地址错误异常（加载或者取指令）
5	AdES	地址错误异常（存储）
6	IBE	取指令的总线错误
7	DBE	加载数据或者存储数据的总线错误
8	Sys	系统调用异常
9	Bp	断点异常
10	RI	保留指令异常
11	CpU	没有实现的协处理器
12	Ov	算术溢出异常
13	Tr	陷阱
15	FPE	浮点



图 A-7-2 原因寄存器

异常和中断导致 MIPS 处理器跳转到一段代码，地址是  $80000180_{16}$ （在核心态，而不是在用户的地址空间），被称作异常处理代码。这个代码检查异常的原因，而且跳转到操作系统的一个合适的点。操作系统对一个异常会做出以下的响应：结束一个引起异常的进程或者执行一些动作。进程所引起的错误，例如执行一个没有被实现的指令，就会被操作系统终止。另一方面，别的异常，例如送给操作系统的来自进程的缺页错误就是要执行一个服务，即从磁盘取回一个页。操作系统处理这些请求，然后恢复这个发出请求的进程。最后一种类型的异常是外部设备发出的中断。这些通常会导致操作系统将数据搬运到 I/O，或者从 I/O 把数据搬运回来，然后恢复被中断的进程。

下面例子中的代码是一个简单异常处理程序，它启动一个为每个异常打印消息的程序（但不是中断）。这个代码与 SPIM 仿真器使用的异常处理 (exceptions.s) 相似。

A-35

### 01 例题·异常处理

异常处理程序首先保存寄存器 \$at，这个符号在处理程序代码的伪代码中被使用，然后保存 \$a0 和 \$a1，这两个值之后将被用来传递参数。异常处理程序不能在堆栈中保存这些寄存器的旧值，作为一个一般的程序，因为异常产生的原因可能是一个内存引用在堆栈指针中使用了一个坏的值（如 0）。相反，异常处理程序在一个异常处理寄存器 (\$k1，因为不使用 \$at，它不能访问内存) 以及两个内存地址 (save0 和 save1) 中保存这些寄存器的值。如果异常处理程序本身可以被中断，两个地址可能不充分，因为第二个异常可能会改写这些第一个异常保存的值。然而，在允许中断之前，这个简单的异常处理程序结束运行，所以这些问题不会出现。

```
.ktext 0x80000180
mov $k1, $at      # Save $at register
sw $a0, save0    # Handler is not re-entrant and can't use
sw $a1, save1    # stack to save $a0, $a1
                  # Don't need to save $k0/$k1
```

异常处理程序然后将原因寄存器和 EPC 寄存器保存到 CPU 的寄存器中。原因寄存器和 EPC 寄存器不是 CPU 寄存器组的一个部分。相反，它们是协处理器 0 的寄存器，协处理器是 CPU 处理异常的一个部分。指令 mfc0 \$k0, \$13 将协处理器 0 的寄存器 13（原因寄存器）保存到 CPU 的寄存器 \$k0。注意到异常处理不需要保存这些寄存器 \$k0 和 \$k1，因为用户程序不被认为会使用这些寄存器。异常处理程序使用来自原因寄存器的值来测试异常是否被一个中断所引起（参见前面的表）。如果是这样，则异常就会被忽略。如果异常不是中断，程序就会调用 print\_excp 来打印一条信息。

A-36

```
mfc0    $k0, $13      # Move Cause into $k0
srl     $a0, $k0, 2      # Extract ExcCode field
andi   $a0, $a0, 0xf
bgtz  $a0, done        # Branch if ExcCode is Int (0)
mov    $a0, $k0          # Move Cause into $a0
mfco   $a1, $14          # Move EPC into $a1
jal    print_excp      # Print exception error message
```

在返回之前，异常处理程序清除原因寄存器；重设状态寄存器，目的是为了使得中断允许。而且清除 EXL 位，这使得后续的异常来修改 EPC 寄存器；而且恢复寄存器 \$a0、\$a1 和 \$at。然后执行 eret 指令（异常返回），这个指令返回 EPC 所指向的指令。这个异常处理程序返回到引起异常的指令的后面的那条指令。所以不会重新执行那条错误的指令，而且不会再引起异常。

```

done:    mfc0      $k0, $14      # Bump EPC
        addiu     $k0, $k0, 4       # Do not re-execute
                                    # faulting instruction
        mtc0      $k0, $14      # EPC

        mtc0      $0, $13      # Clear Cause register

        mfc0      $k0, $12      # Fix Status register
        andi     $k0, 0xffffd   # Clear EXL bit
        ori      $k0, 0x1       # Enable interrupts
        mtc0      $k0, $12

        lw       $a0, save0    # Restore registers
        lw       $a1, save1
        mov      $at, $k1

        eret          # Return to EPC

.kdata
save0: .word 0
save1: .word 0

```

A-37

**01 精解** 在一个实际的 MIPS 处理器中，从异常处理程序返回的过程相当复杂。异常处理程序不能经常跳转到 EPC 的下一条指令。例如，如果引起异常的指令处于分支指令的延迟槽中（见第 4 章），下一条被执行的指令可能就不是内存中的下一条指令。

## A.8 输入和输出

SPIM 仿真一个 I/O 设备：一个内存映射的控制台，在这个控制台中可以读和写字符。当一个程序正在运行时，SPIM 将其终端（一个独立的控制台窗口，是 X-window 版本的 xspim 或者 Windows 版本的 PCSpim）连接到处理器上。运行在 SPIM 上的一个 MIPS 程序可以读取你输入的字符。另外，如果 MIPS 程序可以在终端写字符，字符将出现在 SPIM 的终端或者控制台窗口。这个规则的一个异常是 control-C：这个字符没有被传递到程序中，但是导致 SPIM 停止，而且返回到命令行模式。当程序停止运行（例如，因为你输入 control-C，或者因为程序遇到一个断点）时，终端被重新连接到 SPIM，这样你就可以输入 SPIM 指令了。

为了使用内存映射的 I/O（见下面），spim 或者 xspim 必须使用 -mapped\_i\_o 标志来启动。PCSpim 通过一个命令行标志可以允许内存映射的 I/O。或者通过“设置”对话框来实现。

终端设备由两个独立的单元组成：一个“接收者”和一个“发送者”。接收者读来自键盘的字符。发送者在终端（控制台）显示字符。两个单元完全独立。这意味着，例如，从键盘输入的字符不能自动重复显示。相反，一个程序通过从接收者那里读一个字符，而且将其写到发送者那里。

一个程序控制着一个具有 4 个内存映射的设备寄存器的终端，如图 A-8-1 所示。“内存映射”意味着每个寄存器作为一个特殊的内存地址。一个接收者控制寄存器是在地址  $ffff0000_{16}$ ，实际仅用到它的两个位。位 0 称作“ready”（预备）：如果它是 1，这意味着一个字符从键盘到

达，但是还没有从数据接收器的寄存器中读出。ready 位是只读的：对它的写操作会被忽略。当字符从键盘输入时，ready 位从 0 转换到 1，而且当字符从接收器数据寄存器读取时，ready 位从 1 转换到 0。

A-38

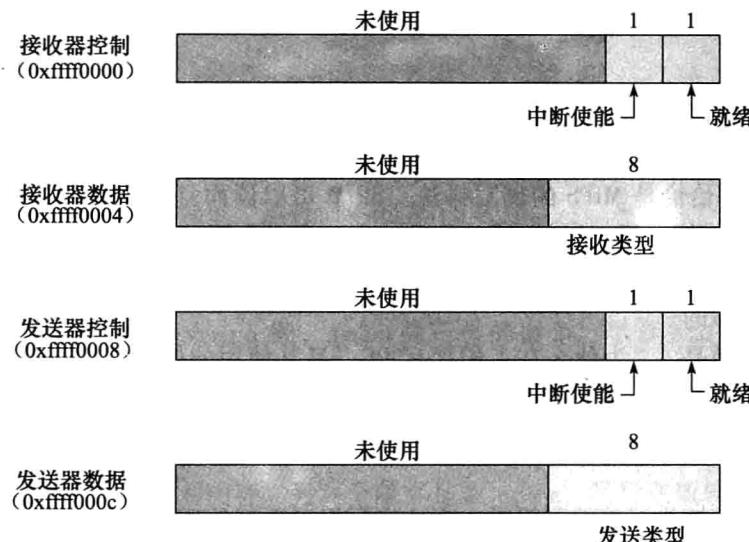


图 A-8-1 终端被 4 个设备寄存器控制着，每个寄存器在给定地址作为一个内存地址。仅仅这些寄存器的一些位实际上被使用。别的位通常被读成 0，对它们的写被忽略

接收器的控制寄存器的第 1 位是键盘“中断允许位”。这个位可以被程序读，也可以被写。中断允许位初始为 0。如果它被程序设置成 1，无论何时字符被输入，终端在硬件级 1 请求一个中断。然而，中断影响到处理器，中断必须在状态寄存器中设置成允许（见 A.7 节）。接收器的控制寄存器别的位都没有被使用。

第二个终端设备寄存器是接收器数据寄存器（在地址  $ffff0004_{16}$ ）。这个寄存器的低 8 位包含着从键盘输入的最后字符，别的位都是 0。这个寄存器是只读而且仅当一个新的字符从键盘被输入时才改变。读取接收器数据寄存器导致接收器控制寄存器的 ready 位复位成 0。如果接收器控制寄存器是 0，这个寄存器中的值没有被定义。

第三个终端设备寄存器是发送者控制寄存器（在地址  $ffff0008_{16}$ ）。当这个寄存器只有低两位被使用。它们的行为和接收器控制寄存器很相似。第 0 位称作“ready”，而且是只读的。如果这个位是 1，发送者准备为输出接受一个新的字符；如果是 0，发送者将仍然忙于写前一个字符。第 1 位是“终端允许位”，而且是可读可写的。如果这个位被设置成 1，无论发送者是否准备好一个新的字符，终端在硬件 0 级请求一个中断，然后 ready 位变成 1。

最后一个设备寄存器是发送者数据寄存器（在地址  $ffff000c_{16}$ ）。当一个值被写入这个地址处时，它的低 8 位（例如，第 2 章的图 2-15 中的一个 ASCII 字符）被发送到控制台。当发送者数据寄存器被写时，发送者控制寄存器的 ready 位被设置成 0。这个位保持为 0，直到字符发送到终端花费了足够的时间；然后 ready 位又一次变成 1。发送者数据寄存器应当仅仅当发送者控制寄存器的 ready 位是 1 的时候可以被写入。如果发送者没有准备好，写入到发送者数据寄存器的数据会被忽略（写入会成功，但是字符没有输出）。

A-39

实际的计算机需要时间来将字符发送到控制台或者终端。这些时间延迟会被 SPIM 仿真器模拟。例如，在发送者开始写一个字符之后，发送者的 ready 位之后不久就变成 0。SPIM 按照

指令的执行来测量时间，而不是按照实际的时间。这就意味着发送者不会变成 ready，直到处理器执行一个固定数量的指令。如果你停止机器，而且查看其 ready 位，它是不会改变的。然而，如果你让机器运行，这个位最终会变成 1。

## A.9 SPIM

SPIM 是一个软件仿真器，它运行为处理器编写的汇编语言程序，实现 MIPS-32 体系结构，尤其体系结构的版本 1 具备固定的内存映射，没有 cache，而且仅有协处理器 0 和协处理器 1。<sup>⊖</sup> SPIM 的名字恰恰是 MIPS 的倒写拼法。SPIM 可以读而且可以立即执行汇编语言文件。

**A-40** SPIM 是一个自含的系统，用于运行 MIPS 程序。它包含一个调试器，并提供一些类似操作系统的服务。SPIM 比实际的计算机要慢得多（100 倍或更多）。然而，它代价小，可用性广泛，是真实硬件所无法比拟的。

一个明显的问题是，“为什么在人们拥有 PC，且其使用的处理器比 SPIM 运行快得多时，却还使用仿真器？”原因之一是这些 PC 中的处理器是 Intel 的 80x86，它们的结构不太规则，而且复杂，难以理解，比 MIPS 处理器编程困难。MIPS 结构可能是一个简单、整洁的 RISC 机器的缩影。

另外，仿真器能够为汇编编程提供一个比实际机器更好的环境，因为它们能检测出更多的错误，提供一个比实际计算机更好的接口。

最后，仿真器是研究计算机和在其上运行的程序的有用工具。因为它们是以软件方式实现的，而不是硅。所以对于添加新指令，构建像多处理器这样的新系统，或者收集数据这样简单的事情，使用仿真器容易验证也容易修改。

### A.9.1 虚拟机的仿真

由于延迟分支、延迟加载、受限制的地址模式等原因，基本的 MIPS 体系结构很难直接编程。这个困难是可忍受的，因为这些计算机被设计用来使用高级语言编程，所以是给编译器，而不是给汇编语言程序员提供接口。编程复杂度的很大一部分是由延迟指令导致的。一个延迟的分支需要两个周期来执行（见第 4 章的 4.5 节和 4.8 节的精解）。在第二个周期，执行那条紧跟分支指令的指令。这条指令能执行有用的工作，而正常情况下该工作可能在分支指令之前已经完成。它可能是没有任何操作的 nop 指令。同样，延迟加载需要两个周期来将一个值从内存中取回，所以紧跟其后的指令无法使用这个值（见第 4 章的 4.2 节）。

MIPS 一般通过汇编语言实现的虚拟机（virtual machine）隐藏复杂度。虚拟机没有延迟的分支、加载，而且具有比实际硬件更丰富的指令集。汇编器将这些指令重新组织（分派）到延迟槽中。虚拟机也提供伪指令，这些指令看起来和汇编语言程序中的真实指令一样。然而，硬件完全不知道这些伪指令，所以汇编器必须将其翻译成实际机器指令的等价序列。例如，当一个寄存器等于 0 或者不等于 0 的时候，MIPS 硬件仅仅提供分支指令。对于其他的条件分支，例如那种当一个寄存器的值比另一个寄存器的值大时就进行分支的类型，分支指令会被综合成两个寄存器的比较，然后当其比较的结果是真（非零）时就执行分支。

**A-41**

虚拟机：一种虚拟计算机，它分支和取数指令没有延迟，且指令集比实际硬件更丰富。

<sup>⊖</sup> SPIM 的早期版本（7.0 以前的版本）实现了 MIPS-1 的体系结构，使用原始的 MIPS R2000 处理器。这个体系结构几乎是 MIPS-32 体系结构的子集，不同在于异常处理的方式。MIPS-32 也引入了将近 60 个新指令，SPIM 支持这些指令。程序可以在 SPIM 的早期版本中运行，而且不使用异常的程序可以不加修改在新版本 SPIM 上运行。使用异常的程序将需要少许的修改。

默认情况下，SPIM 模拟指令集更丰富的虚拟机，因为这是一个对很多程序员来说很有用的机器。然而，SPIM 也能模拟实际的硬件的延迟分支以及延迟取数操作。下面，我们描述虚拟机并且仅提及其实际的硬件没有关系的特性。这样做，我们遵循了 MIPS 的汇编程序员（汇编器）的规程，他们将扩展的机器当成是由硅实现的机器那样使用。

### A.9.2 从 SPIM 开始

本附录的剩余的部分介绍 SPIM 和 MIPS R2000 汇编语言。你不用关注过多的细节，然而，大量的信息很多时候会模糊以下事实：SPIM 是个简单易用的程序。本节我们先以 SPIM 的快速使用教程开始，教会你加载、调试、运行简单的 MIPS 程序。

对于不同类型的计算机系统，SPIM 有几个不同的版本。其中一个经久不变的，是最简单的版本，称为 `spim`，它是运行在控制窗口下的一个命令行驱动程序。它和很多控制台程序一样操作：输入一行文本，按 `return` 键，`spim` 执行你的命令。尽管 `spim` 缺乏精美的界面，但它可以做具有精美界面的同类版本可以做的任何事情。

`spim` 拥有两个界面精美的版本。运行在 UNIX 或者 Linux 系统上的 X-windows 环境下的版本称为 `xspim`。与 `spim` 相比，`xspim` 更易于学习和使用，因为它的指令总是在屏幕上可见的，且持续显示机器的寄存器和内存。另一个版本是 `PCspim`，运行在微软的 Windows 系统下。SPIM 的 UNIX 和 Windows 版本都在本书配套网站上。`xspim`、`pcSpim`、`spim` 的教程和 SPIM 命令行选项都在网站上。

如果你打算在运行微软 Windows 系统的 PC 上运行 SPIM，你应当先阅读网站上的 `PCSpim` 教程。如果你打算在运行 UNIX 或 Linux 的 PC 上运行 SPIM，你应当阅读网站上的 `xspim` 教程。

### A.9.3 令人惊讶的特性

尽管如实地仿真了 MIPS 计算机，但 SPIM 作为一个仿真器，和实际计算机必定是不相同的。最明显的区别是，指令的时序和内存系统不同。SPIM 不模拟 cache 或者存储器的延迟，也不会精确反映浮点操作、乘法或者除法指令的延迟。另外，浮点指令不检测错误条件，而这在实际机器上将导致异常。

另一个令人惊讶的特性（这种情形在真实机器上也会发生）是将伪指令扩展成多条机器指令。当你单步调试或者检查存储器，你所看到的指令和原始的程序不相同。两组指令之间的对应性相当简单，因为 SPIM 并没有为了填充延迟槽而重组指令。

A-42

### A.9.4 字节顺序

处理器能对字中的字节进行编号，这样编号最小的字节不是在最左边就是在最右边。机器使用的该约定称为字节顺序。MIPS 处理器可以依大端字节顺序或者小端字节顺序进行操作。例如，在大端机器下，指令 `byte 0,1,2,3` 将引起一个内存字包含：

字节号			
0	1	2	3

但是在小端机器下，一个字可能包含：

字节号			
3	2	1	0

SPIM 以两种字节顺序操作。SPIM 的字节顺序和运行仿真器的底层机器的字节顺序是一样的。例如，在 Intel 80x86 处理器上，SPIM 是小端，然而在 Macintosh 或者 Sun SPARC 处理器上，SPIM 是大端。

### A.9.5 系统调用

SPIM 通过系统调用 (syscall) 指令提供了一小组类似操作系统的服务。为了请求一个服务，一个程序加载系统调用代码（见图 A-9-1）到寄存器 \$v0，将参数加载到寄存器 \$a0 ~ \$a3（或用于浮点值的 \$f12）。系统调用将返回值放到 \$v0（或用于浮点值的 \$f0）。例如，下面的代码将打印“the answer = 5”：

```

.data
str:
    .asciiz "the answer = "
.text
    li      $v0, 4      # system call code for print_string
    la      $a0, str    # address of string to print
    syscall          # print the string

    li      $v0, 1      # system call code for print_int
    li      $a0, 5      # integer to print
    syscall          # print it

```

A-43

服务	系统调用代码	参数	结果
print_int	1	\$a0 = integer	
print_float	2	\$f12 = float	
print_double	3	\$f12 = double	
print_string	4	\$a0 = string	
read_int	5		integer (in \$v0)
read_float	6		float (in \$f0)
read_double	7		double (in \$f0)
read_string	8	\$a0 = buffer, \$a1 = length	
sbrk	9	\$a0 = amount	address (in \$v0)
exit	10		
print_char	11	\$a0 = char	
read_char	12		char (in \$v0)
open	13	\$a0 = filename (string), \$a1 = flags, \$a2 = mode	file descriptor (in \$a0)
read	14	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars read (in \$a0)
write	15	\$a0 = file descriptor, \$a1 = buffer, \$a2 = length	num chars written (in \$a0)
close	16	\$a0 = file descriptor	
exit2	17	\$a0 = result	

图 A-9-1 系统服务

给 print\_int 系统调用传递一个整数并在终端上打印出来。print\_float 打印一个单精度浮点数；print\_double 打印出来一个双精度数；而给 print\_string 传递一个指向空的终止数据串的指针，这个字符串写到终端上。

系统调用 read\_int、read\_float 以及 read\_double 来读取一个完整的输入行，并包

含一个新行。数字后面的字符串将被忽略。`read_string` 具有和 UNIX 库例程 `fgets` 相同的语义。它将读取的  $n - 1$  个字符存到缓冲区，并使用一个空字节作为结束符。如果当前行中的字符数少于  $n - 1$  个，`read_string` 将读取到新行，并再次使用一个空字节作为结束符。

A-44

**警告：** 使用系统调用从终端读取数据的程序不应当使用内存映射的 I/O (见 A.8 节)。

`sbrk` 返回一个指向包含  $n$  个额外字节块的存储器指针。`exit` 可以终止 SPIM 正在执行的程序。`exit2` 终止 SPIM 程序，并且当 SPIM 仿真器终止时，传递给 `exit2` 的参数将变成返回值。

`print_char` 和 `read_char` 分别读和写单个字符。`open`、`read`、`write` 和 `close` 是 UNIX 的标准库调用。

## A.10 MIPS R2000 汇编语言

MIPS 处理器由整型处理单元 (CPU) 和一系列协处理器 (用于执行辅助工作或诸如浮点等其他数据类型的操作) 组成 (见图 A-10-1)。SPIM 可模拟两个协处理器。协处理器 0 用于处理异常和中断。协处理器 1 是浮点运算单元。SPIM 模拟本单元的大多数功能。

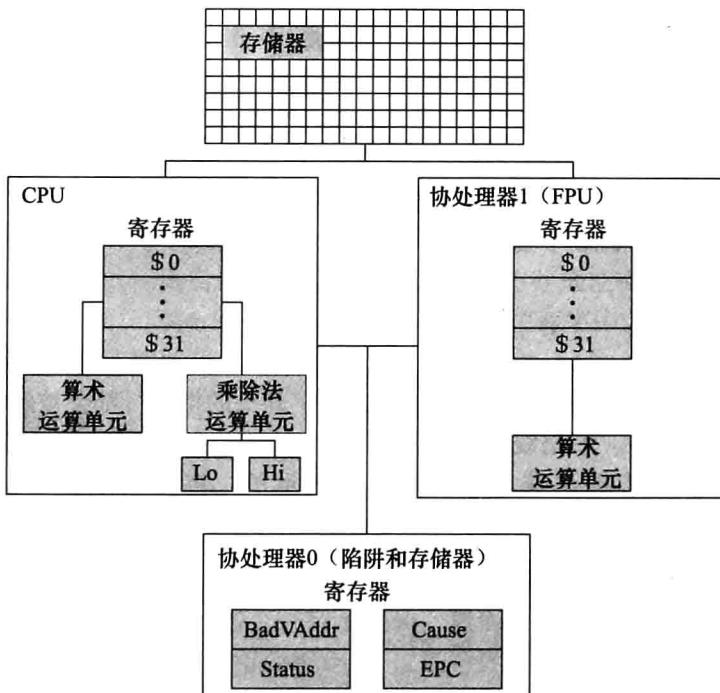


图 A-10-1 MIPS R2000 CPU 和 FPU

### A.10.1 寻址模式

MIPS 采取加载和存储体系结构，也就是说，只有加载和存储指令访问存储器。计算指令只对寄存器中的值进行处理。机器本身只提供一种存储器寻址模式：`c(rx)`。它把立即数 `c` 和寄存器 `rx` 的值相加作为地址。虚拟机则为加载和存储指令提供了以下几种寻址方式：

格式	地址计算
(寄存器)	寄存器内容
立即数	立即数
立即数 (寄存器)	立即数 + 寄存器内容

(续)

格式	地址计算
标识	地址标识
标识 ± 立即数	地址标识 + 或 - 立即数
标识 ± 立即数 (寄存器)	地址标识 + 或 - (立即数 + 寄存器内容)

A-45 大多数加载和存储指令只对对齐数据进行处理。以字节为单位，当数据所在的存储器地址是其大小的整数倍时，我们就说数据是对齐的。因此，半字对象必须存放在偶地址，而全字对象必须存放在 4 的整数倍的地址。但是，MIPS 还提供了另外一些指令，可以对非对齐数进行操作（如 lw1, lwr, swl 和 swr）。

**01 精解** MIPS 汇编器（SPIM 也一样）通过在对数据存取之前产生一条或多条指令来计算复杂的地址，因此它也支持一些复杂的寻址模式。例如，假设标识 table 指向存储器地址 0x10000004，并且程序包含一条这样的指令：

```
ld $a0, table + 4($a1)
```

汇编器会将这条指令转化为下面三条指令：

```
lui $at, 4096  
addu $at, $at, $a1  
lw $a0, 8($at)
```

第一条将标记地址的高位送入寄存器 \$at（该寄存器是汇编器为自己保留的）。第二条将寄存器 \$a1 的内容加到标识的局部地址上。最后，加载指令用硬件寻址模式将标识地址的低位和寄存器 \$at 中相对原始指令的偏移量相加。

## A. 10.2 汇编语法

汇编文件中的注释行以“#”开始。所有以“#”开头的指令行都会被忽略。

标识符由字母、数字、下划线（\_）和点（.）构成，但不能以数字开头。指令操作码是一些保留字，不能用作标识符。标识是这样表示的：将其放在行首，后跟冒号（:）。例如：

```
.data  
item: .word 1  
.text  
.globl main      # Must be global  
main: lw         $t0, item
```

数值默认是十进制。如果数值以 0x 开始，则表明它们是十六进制数。因此，256 和 0x100 所表示的数值是相同的。

字符串用双引号（"）括起来。字符串中的特殊字符遵从 C 语言规范：

- 换行 \n
- 制表 \t
- 引号 \"

SPIM 还支持一些 MIPS 汇编指令：

.align n	将数据以 $2^n$ 字节分界。例如，.align 2 将数据以字为单位分界；.align 0 关闭.half、.word、.float 和.double 的自动分界方法，直到出现.data 或.kdata 为止。
.ascii str	将字符串 str 存入主存中，但不以空字符结束。
.asciiz str	将字符串 str 存入主存，并以空字符结束。

.byte b1, …, bn	将 n 个值存入主存的连续字节中。
.data <addr>	将后续项存入数据段中。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。
.double d1, …, dn	将 n 个双精度浮点数存入连续的主存单元。
.extern sym size	声明存储在 sym 中大小为 size 字节的全局变量。该指令允许汇编器将数据存放到数据段中，这样可以用 \$gp 寄存器快速存取。
.float f1, …, fn	将 n 个单精度浮点数存入连续的主存单元。
.globl sym	声明 sym 是全局标记，可以在其他文件中引用。
.half h1, …, hn	将 n 个 16 位数据存入连续的主存单元。
.kdata <addr>	将后续数据项存入核心数据段中。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。
.ktext <addr>	将后续项放入核心代码段。在 SPIM 中，这些后续项只能是指令或字（参看下面的 .word 指令）。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。
.set noat 和 .set at	前一指令阻止 SPIM 对后续指令中使用 \$at 寄存器的警告，后一指令恢复这种警告。由于伪指令展开成指令时会用到寄存器 \$at，程序员必须谨慎使用寄存器 \$at。
.space n	在当前段分配 n 字节（SPIM 中则必须为数据段）。
.text <addr>	将后续项送入用户代码段中。在 SPIM 中，这些后续项只能是指令或字（参看下面的 .word 指令）。如果给出了可选参数 addr，则后续项存入以 addr 开始的主存地址中。
.word w1, …, wn	将 n 个 32 位数据存入连续的主存字中。
SPIM 不区分数据段的不同部分 (.data, .rdata 和 .sdata)。	A-48

### A. 10.3 MIPS 指令编码

图 A-10-2 描述了 MIPS 指令是如何以二进制数进行编码的。每一列包含指令字段（邻接的一组二进制位）的编码。左边界上的数字是对应字段的值。例如，操作码 j 在操作码字段的值为 2。每列顶上的文字定义了一个字段，并且指出了占用指令中的哪些位。例如，op 字段对应指令中的 26~31 位。该字段对大多数指令进行了编码。然而，有些指令组用到了附加字段以区别相关的指令。例如，不同的浮点数指令用 0~5 位进行区别。第一列的箭头表明哪些操作码用到了这些附加字段。

### A. 10.4 指令格式

本附录的剩余部分将对由 MIPS 硬件实现的指令和 MIPS 汇编器实现的伪指令进行描述。这两种指令很容易区分。实际指令的字段用对应的二进制来表示。例如：

加法操作（带溢出位）

add rd, rs, rt	0	rs	rt	rd	0	0x20
	6	5	5	5	5	6

add 指令由 6 个字段组成。字段的长度标在字段下面。该指令由 6 位 0 开始。寄存器标识符以 r 开始，因此接下来的字段是称为 rs 的 5 位寄存器标识符。它与本行左边汇编代码中的第二个参数相同。另一个常用字段是 imm<sub>16</sub>，它是一个 16 位立即数。

A-49

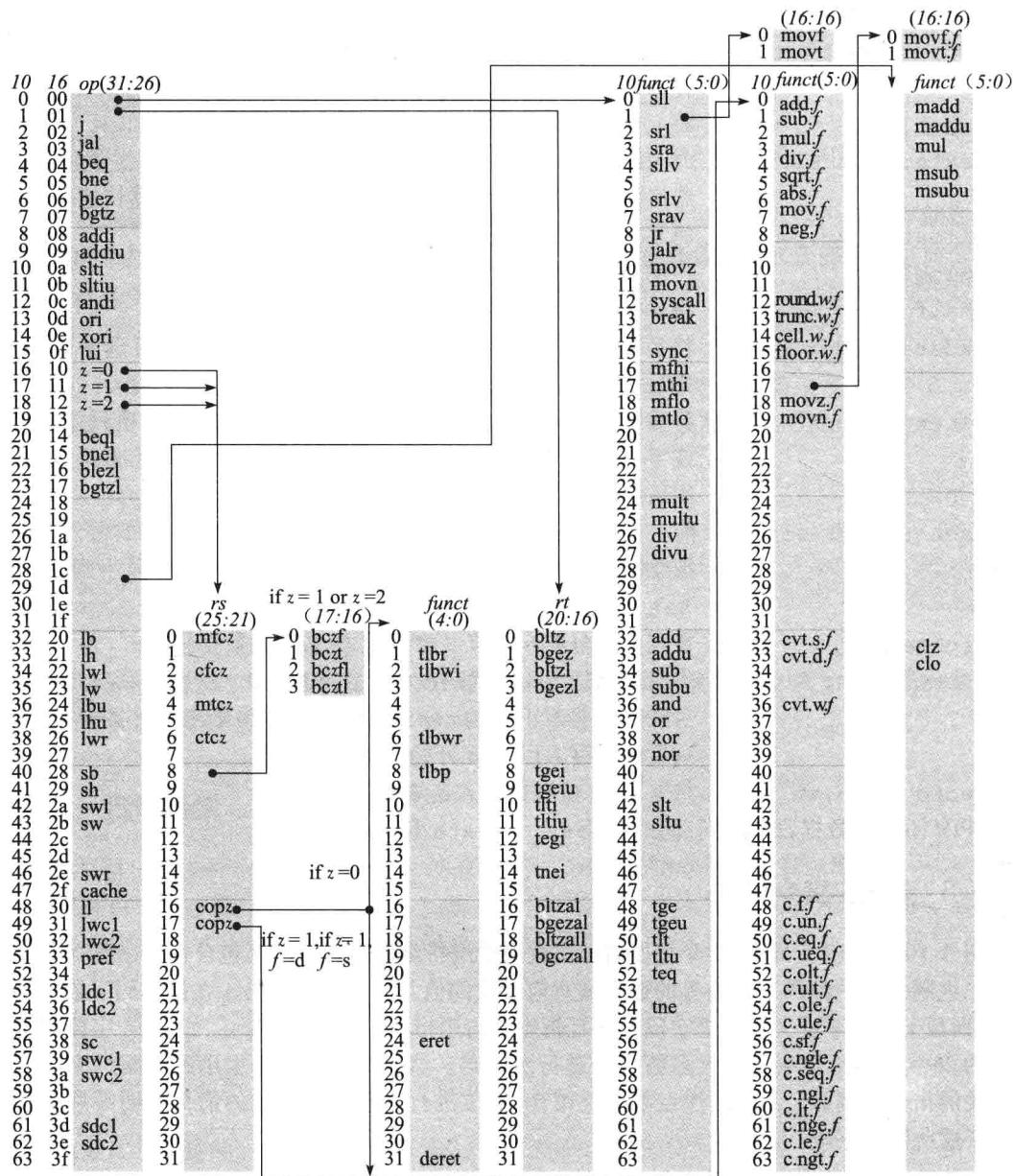


图 A-10-2 MIPS 操作码图。每个字段的数值在它的左侧显示。第一列、第二列是第三列中操作符字段 (31~26位) 对应的十进制值和十六进制值。该操作符字段能表达除了6个操作数(0, 1, 16, 17, 18, 19)以外的任何MIPS操作。这些操作由其他字段确定,由指针进行识别。如果  $rs = 16$ ,  $op = 17$ , 最后的字段 (funct) 用 “f” 表示 “s”; 如果  $rs = 17$ ,  $op = 17$ , 则 “f” 表示 “d”。如果  $op = 16, 17, 18, 19$ , 则第二列 (rs) 用 “z” 分别表示: “0”, “1”, “2”, “3”。如果  $rs = 16$ , 则操作由别处定义; 如果  $z = 0$ , 则操作在第四个字段中定义 (4~0位); 如果  $z = 1$ , 则操作在最后的字段中, 且  $f = s$ 。如果  $rs = 17$  且  $z = 1$ , 则操作在最后的字段中, 且  $f = d$ 。

伪指令大体上遵循这些约定,但省略了指令编码信息。例如:

乘操作 (不带溢出位)

mul rdest, rsrc1, src2

伪指令

在伪指令中，rdest 和 rsrc1 表示寄存器，而 src2 表示寄存器或立即数。通常情况下，汇编器和 SPIM 将一条通用的指令格式（例如，add \$v1, \$a0, 0x55）转化为特定的形式（例如，addi \$v1, \$a0, 0x55）。

## A. 10.5 算术和逻辑指令

### 绝对值

abs rdest, rsrc 伪指令

将寄存器 rsrc 的值求绝对值再存入寄存器 rdest 中。

#### 加法（带溢出位）

add rd, rs, rt	0	rs	rt	rd	0	0x20
	6	5	5	5	5	6

#### 加法（不带溢出位）

addu rd, rs, rt	0	rs	rt	rd	0	0x21
	6	5	5	5	5	6

将寄存器 rs 和 rt 的和存入寄存器 rd 中。

#### 立即数加（带溢出位）

addi rt, rs, imm	8	rs	rt	imm	
	6	5	5	16	

#### 立即数加（不带溢出位）

addiu rt, rs, imm	9	rs	rt	imm	
	6	5	5	16	

将寄存器 rs 与立即数之和存入寄存器 rt 中。

### 逻辑与

and rd, rs, rt	0	rs	rt	rd	0	0x24
	6	5	5	5	5	6

将寄存器 rs 与 rt 进行逐位逻辑与，结果存入寄存器 rd。

### 立即数与

andi rt, rs, imm	0xc	rs	rt	imm	
	6	5	5	16	

将寄存器 rs 同立即数进行逐位逻辑与，结果存入寄存器 rt。

### 统计起始为 1 的个数

clz rd, rs	0x1c	rs	0	rd	0	0x21
	6	5	5	5	5	6

### 统计起始为 0 的个数

clz rd, rs	0x1c	rs	0	rd	0	0x20
	6	5	5	5	5	6

将寄存器 rs 中数据起始为 1 (0) 的个数存入寄存器 rd，如果字中都是 1 (0)，则结果为 32。

### 除法（带溢出位）

div rs, rt	0	rs	rt	0	0x1a
	6	5	5	10	6

### 除法（不带溢出位）

divu rs, rt	0	rs	rt	0	0x1b
	6	5	5	10	6

寄存器 rs 被寄存器 rt 除，将商存入寄存器 lo，将余数存入寄存器 hi。如果其中有某个操作数是负数，则余数取决于运行 SPIM 的计算机系统，而与 MIPS 体系结构无关。

A-50  
A-51

A-52

**除法 (带溢出位)**

div rdest, rsrcl, src2 伪指令

**除法 (不带溢出位)**

divu rdest, rsrcl, src2 伪指令

将寄存器 rsrcl 和 src2 的商存入寄存器 rdest。

**乘法**

mult rs, rt	0	rs	rt	0	0x18
	6	5	5	10	6

**无符号数乘法**

multu rs, rt	0	rs	rt	0	0x19
	6	5	5	10	6

将寄存器 rs 和 rt 的数据相乘，乘积的低位字和高位字分别存入寄存器 lo 和 hi。

**乘积 (不带溢出位)**

mul rd, rs, rt	0x1c	rs	rt	rd	0	2
	6	5	5	5	5	6

将 rs 和 rt 乘积的低 32 位存入寄存器 rd 中。

**乘积 (带溢出位)**

mulo rdest, rsrcl, src2 伪指令

**无符号数相乘 (带溢出位)**

mulou rdest, rsrcl, src2 伪指令

A-53

将寄存器 rsrcl 和 src2 的乘积结果的低 32 位存入寄存器 rdest。

**乘加**

madd rs, rt	0x1c	rs	rt	0	0
	6	5	5	10	6

**无符号乘加**

maddu rs, rt	0x1c	rs	rt	0	1
	6	5	5	10	6

将寄存器 rs 和 rt 的乘积所得的 64 位结果与连接寄存器 lo 和 hi 中的 64 位值相加。

**乘减**

msub rs, rt	0x1c	rs	rt	0	4
	6	5	5	10	6

**无符号乘减**

msubu rs, rt	0x1c	rs	rt	0	5
	6	5	5	10	6

将寄存器 rs 和 rt 的乘积所得的 64 位结果与连接寄存器 lo 和 hi 中的 64 位值相减。

**求相反数 (带溢出位)**

neg rdest, rsrcl 伪指令

**求相反数 (不带溢出位)**

negu rdest, rsrcl 伪指令

将寄存器 rsrcl 的相反数存入寄存器 rdest。

**异或**

nor rd, rs, rt	0	rs	rt	rd	0	0x27
	6	5	5	5	5	6

将寄存器 rs 和 rt 的异或结果存入寄存器 rd。

A-54

**取反****not rdest, rsrc** 伪指令

将寄存器 rsrc 逐位取反存入寄存器 rdest。

**逻辑或****or rd, rs, rt**

0	rs	rt	rd	0	0x25
6	5	5	5	5	6

将寄存器 rs 和 rt 按位逻辑或的结果存入寄存器 rd。

**逻辑或 (立即数)****ori rt, rs, imm**

Oxd	rs	rt	imm
6	5	5	16

将寄存器 rs 和 0 扩展立即数按位逻辑或的结果存入寄存器 rt。

**求余数****rem rdest, rsrc1, rsrc2**

伪指令

**求无符号数的余数****remu rdest, rsrc1, rsrc2**

伪指令

寄存器 rsrc1 被寄存器 rsrc2 除，将余数存入寄存器 rdest。注意，如果有某个操作数是负数，则余数取决于运行 SPIM 的计算机系统，而与 MIPS 体系结构无关。

**逻辑左移****sll rd, rt, shamt**

0	rs	rt	rd	shamt	0
6	5	5	5	5	6

**逻辑左移变量****sllv rd, rt, rs**

0	rs	rt	rd	0	4
6	5	5	5	5	6

A-55

**算术右移****sra rd, rt, shamt**

0	rs	rt	rd	shamt	3
6	5	5	5	5	6

**算术右移变量****srav rd, rt, rs**

0	rs	rt	rd	0	7
6	5	5	5	5	6

**逻辑右移****srl rd, rt, shamt**

0	rs	rt	rd	shamt	2
6	5	5	5	5	6

**逻辑右移变量****srlv rd, rt, rs**

0	rs	rt	rd	0	6
6	5	5	5	5	6

由立即数 shamt 或寄存器 rs 指定寄存器 rt 的左移或右移位数，并将结果存入寄存器 rd。注意，变量 rs 被 sll、sra 和 srl 所忽略。

**循环左移****rol rdest, rsrc1, rsrc2**

伪指令

**循环右移****ror rdest, rsrc1, rsrc2**

伪指令

将寄存器 rsrc1 左移或右移由 rsrc2 指定的位数，然后将结果存入寄存器 rdest。

**减法 (带溢出位)****sub rd, rs, rt**

0	rs	rt	rd	0	0x22
6	5	5	5	5	6

A-56

### 减法（不带溢出位）

subu rd, rs, rt	0	rs	rt	rd	0	0x23
	6	5	5	5	5	6

将寄存器 rs 减去寄存器 rt 并将结果存入寄存器 rd。

### 异或

xor rd, rs, rt	0	rs	rt	rd	0	0x26
	6	5	5	5	5	6

将寄存器 rs 和 rt 按位逻辑异或的结果存入寄存器 rd。

### 异或（同立即数）

xori rt, rs, imm	Oxe	rs	rt	Imm		
	6	5	5	16		

将寄存器 rs 和 0 扩展立即数按位逻辑异或的结果存入寄存器 rt。

## A. 10.6 常数操作指令

### 立即数高位取指令

lui rt, imm	Oxf	0	rt	imm		
	6	5	5	16		

将立即数 imm 的低半字位存入寄存器 rt 的高半字位地址，并将寄存器的低位值置为 0。

### 取立即数

li rdest, imm 伪指令

将立即数 imm 存入寄存器 rdest。

## A. 10.7 比较指令

### 小于指令

slt rd, rs, rt	0	rs	rt	rd	0	0x2a
	6	5	5	5	5	6

### 小于无符号数指令

sltu rd, rs, rt	0	rs	rt	rd	0	0x2b
	6	5	5	5	5	6

若寄存器 rs 比 rt 小，则将寄存器 rd 置为 1；否则，将 rd 置为 0。

### 小于立即数

slti rt, rs, imm	Oxa	rs	rt	imm		
	6	5	5	16		

### 小于立即数（无符号数）

sltiu rt, rs, imm	Oxb	rs	rt	imm		
	6	5	5	16		

若寄存器 rs 比符号扩展立即数小，则将寄存器 rt 置为 1；否则，将 rt 置为 0。

### 等于

seq rdest, rsrcl, rsrcc 伪指令

若寄存器 rsrcl 与寄存器 rsrcc 的数值相等，则将寄存器 rdest 置为 1；否则，将 rdest 置为 0。

### 大于等于

sge rdest, rsrcl, rsrcc 伪指令

**大于等于无符号数**

`sgeu rdest, rsrc1, rsrc2` 伪指令

若寄存器 `rsrc1` 大于等于寄存器 `rsrc2` 的值，则将寄存器 `rdest` 置为 1；否则，将 `rdest` 置为 0。

**大于**

`sgt rdest, rsrc1, rsrc2` 伪指令

A-58

**大于无符号数**

`sgtu rdest, rsrc1, rsrc2` 伪指令

如果寄存器 `rsrc1` 的值大于 `rsrc2` 的值，那么令寄存器 `rdest` 的值为 1，否则为 0。

**小于等于无符号数**

`sle rdest, rsrc1, rsrc2` 伪指令

**小于等于无符号数**

`sieu rdest, rsrc1, rsrc2` 伪指令

如果寄存器 `rsrc1` 的值小于等于 `rsrc2` 的值，那么令寄存器 `rdest` 的值为 1，否则为 0。

**不等**

`sne rdest, rsrc1, rsrc2` 伪指令

如果寄存器 `rsrc1` 的值不等于 `rsrc2` 的值，那么令寄存器 `rdest` 的值为 1，否则为 0。

## A. 10.8 分支指令

分支指令使用了一个有符号的 16 位指令偏移域；因此，指令跳转的范围可以是向前的  $2^{15} - 1$  条指令（非字节），或者向后的  $2^{15}$  条指令。跳转指令包含了一个 26 位的地址域。在实际的 MIPS 处理器中，分支指令是延迟的分支，直到分支指令后面的指令（延迟槽）执行后，才能进行控制转移（见第 4 章）。当分支发生时，由于需要计算相关的延迟槽指令（`PC + 4`）的地址，因此延迟的分支会影响偏移量的计算。除非明确指定 `-bare` 或者 `-delayed_branch` 的标志，否则 SPIM 不模拟延迟槽。

在汇编语言中，偏移量并不具体指定为数字。而是用一个指向标记的指令，并用汇编器计算出分支指令和目标指令之间的距离。

在 MIPS-32 中，所有实际的（不是伪的）条件分支指令都有相似的变体（例如，与 `beq` 相似的变体是 `beql`），如果分支没有发生，那么分支延迟槽中的指令就不能执行。不要使用这些指令，在后续的体系结构版本中，它们可能将被删除。SPIM 实现了这些指令，但并没有做深入讨论。

A-59

**分支指令**

`b label` 伪指令

无条件转移到标记的指令。

**分支协处理器假**

<code>bclf cc label</code>	0x11	8	cc	0	Offset
	6	5	3	2	16

**分支协处理器真**

<code>bclt cc label</code>	0x11	8	cc	1	Offset
	6	5	3	2	16

如果浮点协处理器条件标记 `cc` 为假（真），条件转移的指令数由偏移量所指定。如果 `cc` 被指令所忽略，条件码标记为 0。

### 相等分支

beq rs, rt, label	4 6	rs 5	rt 5	Offset 16
-------------------	--------	---------	---------	--------------

如果寄存器值 rs 和 rt 相等，条件转移的指令数由偏移量所指定。

### 大于等于 0 分支

bgez rs, label	1 6	rs 5	1 5	Offset 16
----------------	--------	---------	--------	--------------

A-60 如果寄存器 rs 的值大于等于 0，条件转移的指令数由偏移量所指定。

### 大于等于 0 分支并链接

bgezal rs, label	1 6	rs 5	0x11 5	Offset 16
------------------	--------	---------	-----------	--------------

如果寄存器 rs 的值大于等于 0，条件转移的指令数由偏移量所指定，并将下一条指令地址保存在寄存器 31 中。

### 大于 0 分支

bgtz rs, label	7 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

如果寄存器 rs 的值大于 0，条件转移的指令数由偏移量所指定。

### 小于等于 0 分支

blez rs, label	6 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

如果寄存器 rs 的值小于等于 0，条件转移的指令数由偏移量所指定。

### 小于 0 分支并链接

bltzal rs, label	1 6	rs 5	0x10 5	Offset 16
------------------	--------	---------	-----------	--------------

如果寄存器 rs 的值小于 0，条件转移的指令数由偏移量所指定，并将下一条指令地址保存在寄存器 31 中

### 小于 0 分支

bltz rs, label	1 6	rs 5	0 5	Offset 16
----------------	--------	---------	--------	--------------

A-61 如果寄存器 rs 的值小于 0，条件转移的指令数由偏移量所指定。

### 不相等分支

bne rs, rt, label	5 6	rs 5	rt 5	Offset 16
-------------------	--------	---------	---------	--------------

如果寄存器 rs 与 rt 中的值不相等，条件转移的指令数由偏移量所指定。

### 等于 0 分支

beqz rsrc, label 伪指令

如果 rsrc 等于 0，条件转移到标记的指令那里。

### 大于等于分支

bge rsrc1, rsrc2, label 伪指令

### 大于等于无符号数分支

bgeu rsrc1, rsrc2, label 伪指令

如果寄存器 rsrc1 的值大于等于 rsrc2 的值，条件转移到标记的指令那里。

### 大于分支

bgt rsrc1, rsrc2, label 伪指令

**大于无符号数分支**

bgtu rsrc1, src2, label 伪指令

如果寄存器 rsrc1 的值大于 src2 的值，条件转移到标记的指令那里。

**小于等于分支**

ble rsrc1, src2, label 伪指令

A-62

**小于等于无符号数分支**

bleu rsrc1, src2, label 伪指令

如果寄存器 rsrc1 的值小于等于 src2 的值，条件转移到标记的指令那里。

**小于分支**

blt rsrc1, rsrc2, label 伪指令

**小于无符号数分支**

bltu rsrc1, rsrc2, label 伪指令

如果寄存器 rsrc1 的值小于 rsrc2 的值，条件转移到标记的指令那里。

**不等于 0 分支**

bnez rsrc, label 伪指令

如果寄存器 rsrc 的值不等于 0，条件转移到标记的指令那里。

**A. 10.9 跳转指令****跳转**

j target	2	target
	6	26

无条件跳转到目标指令。

**跳转并链接**

jal target	3	target
	6	26

无条件跳转到目标指令，并将下一条指令地址保存到 \$ra 中。

A-63

**跳转并链接到寄存器**

jalr rs, rd	0	rs	0	rd	0	9
	6	5	5	5	5	6

无条件跳转到由寄存器 rs 指定的指令（指令地址在寄存器 rs 中），并将下一条指令地址保存在寄存器 rd 中（默认为 31）。

**寄存器跳转**

jr rs	0	rs	0	8
	6	5	15	6

无条件跳转到由寄存器 rs 指定的指令。

**A. 10.10 陷阱指令****等于陷阱**

teq rs, rt	0	rs	rt	0	0x34
	6	5	5	10	6

如果寄存器 rs 的值等于寄存器 rt 的值，引发陷阱异常。

**等于立即数陷阱**

teqi rs, imm	1	rs	0xc	imm
	6	5	5	16

如果寄存器 rs 的值等于符号扩展值——imm，引发陷阱异常。

#### 不等于陷阱

teq rs, rt	0 6	rs 5	rt 5	0 10	0x36 6
------------	--------	---------	---------	---------	-----------

如果寄存器 rs 的值不等于寄存器 rt 的值，引发陷阱异常。

#### 不等于立即数陷阱

teqi rs, imm	1 6	rs 5	0xe 5	imm 16
--------------	--------	---------	----------	-----------

A-64 如果寄存器 rs 的值不等于符号扩展值——imm，引发陷阱异常。

#### 大于等于陷阱

tge rs, rt	0 6	rs 5	rt 5	0 10	0x30 6
------------	--------	---------	---------	---------	-----------

#### 大于等于无符号数陷阱

tgeu rs, rt	0 6	rs 5	rt 5	0 10	0x31 6
-------------	--------	---------	---------	---------	-----------

如果寄存器 rs 的值大于或等于寄存器 rt 的值，引发陷阱异常。

#### 大于等于立即数陷阱

tgei rs, imm	1 6	rs 5	8 5	imm 16
--------------	--------	---------	--------	-----------

#### 大于等于无符号立即数陷阱

tgeiu rs, imm	1 6	rs 5	9 5	imm 16
---------------	--------	---------	--------	-----------

如果寄存器 rs 的值大于等于符号扩展值——imm，引发陷阱异常。

#### 小于陷阱

tlr rs, rt	0 6	rs 5	rt 5	0 10	0x32 6
------------	--------	---------	---------	---------	-----------

#### 小于无符号数陷阱

tlru rs, rt	0 6	rs 5	rt 5	0 10	0x33 6
-------------	--------	---------	---------	---------	-----------

如果寄存器 rs 的值小于寄存器 rt 的值，引发陷阱异常。

#### 小于立即数陷阱

tlri rs, imm	1 6	rs 5	a 5	imm 16
--------------	--------	---------	--------	-----------

#### 小于无符号立即数陷阱

tlriu rs, imm	1 6	rs 5	b 5	imm 16
---------------	--------	---------	--------	-----------

如果寄存器 rs 的值小于符号扩展值——imm，引发陷阱异常。

A-65

## A. 10.11 取数指令

### 取地址

la rdest, address 伪指令

将计算的地址——不是地址中的内容——保存到寄存器 rdest 中。

### 取字节

lb rt, address	0x20 6	rs 5	rt 5	Offset 16
----------------	-----------	---------	---------	--------------

**取字节 (无符号)**

lbu rt, address	0x24	rs	rt	Offset
	6	5	5	16

将地址 address 中的字节内容存入寄存器 rt 中，字节由 1b 符号扩展，而不是由 lbu。

**取半字**

lh rt, address	0x21	rs	rt	Offset
	6	5	5	16

**取半字 (无符号)**

lhu rt, address	0x25	rs	rt	Offset
	6	5	5	16

将地址 address 中 16 位数值（半字）存入寄存器 rt 中，半字由 1h 符号扩展，而不是由 lhu。

A-66

**取字**

lw rt, address	0x23	rs	rt	Offset
	6	5	5	16

将地址 address 中 32 位数值（字）存入寄存器 rt 中。

**协处理器 1 取字**

lwcl ft, address	0x31	rs	rt	Offset
	6	5	5	16

将地址 address 中的字以浮点单元的形式存入寄存器 ft 中。

**取左半字**

lwl rt, address	0x22	rs	rt	Offset
	6	5	5	16

**取右半字**

lwr rt, address	0x26	rs	rt	Offset
	6	5	5	16

将可能非对齐地址 address 中值的左（右）半字存入寄存器 rt 中。

**取双字**

ld rdest, address      伪指令

将地址 address 对应的 64 位数值存入寄存器 rdest 和 rdest + 1 中。

**非对齐地址中取半字**

ulh rdest, address      伪指令

A-67

**非对齐地址中取半字 (无符号)**

ulhu rdest, address      伪指令

将可能非对齐地址 address 中 16 位数值（半字）存入寄存器 rdest 中，半字由 ulh 符号扩展，而不是由 ulhu。

**非对齐地址中取字**

ulw rdest, address      伪指令

将可能非对齐地址 address 中 32 位数值（字）存入寄存器 rdest 中。

**链接取**

ll rt, address	0x30	rs	rt	Offset
	6	5	5	16

将 address 中 32 位数值存入寄存器 rt 中，并且开始执行原子读一修改一写操作。该操作由条件存指令 (sc) 来完成，但如果其他处理器对包含有被取字的块进行写操

作时，该操作将失败。由于 SPIM 不能模拟多处理器，因而条件存操作总是可以成功执行的。

### A. 10. 12 保存指令

#### 存字节

sb rt, address	0x28	rs	rt	Offset
	6	5	5	16

将寄存器 rt 的低字节保存到地址 address 中。

#### 存半字

sh rt, address	0x29	rs	rt	Offset
	6	5	5	16

**A-68** 将寄存器 rt 的低 16 位值（半字）保存到地址 address 中。

#### 存字

sw rt, address	0x2b	rs	rt	Offset
	6	5	5	16

将寄存器 rt 中的字保存到地址 address 中。

#### 协处理器 1 存字

swcl ft, address	0x31	rs	ft	Offset
	6	5	5	16

将浮点协处理器中寄存器 ft 中的值以浮点类型存入地址 address 中。

#### 协处理器 1 存双字

sdcl ft, address	0x3d	rs	ft	Offset
	6	5	5	16

**A-69** 将浮点协处理器中寄存器 ft 和 ft + 1 中的数值以浮点类型存入地址 address 中。寄存器 ft 必须偶数化。

#### 存左半字

swl rt, address	0x2a	rs	rt	Offset
	6	5	5	16

#### 存右半字

swr rt, address	0x2e	rs	rt	Offset
	6	5	5	16

将寄存器 rt 中的左（右）半字保存到可能非对齐地址 address 中。

#### 存双字

sd rsrc, address 伪指令

**A-69** 将寄存器 rsrc 和 rsrc + 1 中的 64 位数值保存到地址 address 中。

#### 非对齐地址中存半字

ush rsrc, address 伪指令

将寄存器 rsrc 中的低 16 位（半字）保存到可能的非对齐地址 address 中。

#### 非对齐地址中存字

usw rsrc, address 伪指令

将寄存器 rsrc 中的字保存到可能的非对齐地址 address 中。

#### 条件存

sc rt, address	0x38	rs	rt	Offset
	6	5	5	16

将寄存器 rt 中的 32 位数值（字）存入内存地址 address 中，并完成原子读一修改一写操作。如果原子操作成功执行，内存中的字被修改，寄存器 rt 的值设置为 1。如果由于其他处理器对包含地址字的块进行写操作而导致原子操作失败，该指令不能修改内存，并将寄存器 rt 的值设置为 0。由于 SPIM 不能模拟多处理器，因而该指令总是可以成功执行的。

### A. 10.13 数据传送指令

#### 传送指令

`move rdest, rsrc` 伪指令

将寄存器 rsr<sub>c</sub> 中的数值传送到寄存器 rdest 中。

从 hi 寄存器传送

<code>mfhi rd</code>	0	0	rd	0	0x10
	6	10	5	5	6

A-70

从 lo 寄存器传送

<code>mflo rd</code>	0	0	rd	0	0x12
	6	10	5	5	6

乘法和除法单元将处理的结果存入 hi 和 lo 这两个额外的寄存器中。这些指令向（从）这些寄存器中传送数据。乘、除、取余伪指令像使用通用寄存器那样使用这些单元，并在计算结束后传送结果。

将寄存器 hi (lo) 中的数值传送到寄存器 rd 中。

传送至 hi 寄存器

<code>mthi rs</code>	0	rs	0	0x11
	6	5	15	6

传送至 lo 寄存器

<code>mtlo rs</code>	0	rs	0	0x13
	6	5	15	6

将寄存器 rs 的值传送到 hi (lo) 寄存器。

从协处理器 0 中传送

<code>mfc0 rt, rd</code>	0x10	0	rt	rd	0
	6	5	5	5	11

从协处理器 1 中传送

<code>mfcl rt, fs</code>	0x11	0	rt	fs	0
	6	5	5	5	11

协处理器有它们自己的寄存器集合。这些指令在协处理器的寄存器和 CPU 寄存器之间传送数据。

将协处理器中寄存器 rd (在 FPU 中是 fs) 的值传送到 CPU 寄存器 rt 中。浮点单元使用协处理器 1。

A-71

从协处理器 1 中传送双字

`mfc1.d rdest, frsrc1` 伪指令

将浮点寄存器 frsrc1 和 frsrc1 + 1 中的值传送到 CPU 寄存器 rdest 和 rdest + 1 中。

传送到协处理器 0

<code>mtc0 rd, rt</code>	0x10	4	rt	rd	0
	6	5	5	5	11

### 传送到协处理器 1

mtc1 rd, fs	0x11	4	rt	fs	0
	6	5	5	5	11

将 CPU 中寄存器 rt 的值传送到协处理器的寄存器 rd 中（或者 FPU 的寄存器 fs 中）。

### 非零条件传送

movn rd, rs, rt	0	rs	rt	rd	0xb
	6	5	5	5	11

如果寄存器 rt 的值不为 0，将寄存器 rs 中的数值传送到寄存器 rd 中。

### 零条件传送

movz rd, rs, rt	0	rs	rt	rd	0xa
	6	5	5	5	11

如果寄存器 rt 的值为 0，将寄存器 rs 中的数值传送到寄存器 rd 中。

### FP 值为假时条件传送

movf rd, rs, cc	0	rs	cc	0	rd	0	1
	6	5	3	2	5	5	6

如果 FPU 条件码标记 cc 为 0，将 CPU 寄存器 rs 中的值传送到寄存器 rd 中。如果 cc 被指

A-72 令忽略，那么条件码标记为 0。

### FP 值为真时条件传送

movt rd, rs, cc	0	rs	cc	1	rd	0	1
	6	5	3	2	5	5	6

如果 FPU 条件码标记 cc 为 1，将 CPU 寄存器 rs 中的值传送到寄存器 rd 中。如果 cc 被指  
令忽略，那么条件码标记为 0。

## A.10.14 浮点运算指令

MIPS 中有专门的浮点协处理器（序号为 1），可以执行单精度浮点数（32 位）和双精度浮点数（64 位）。协处理器有自己的寄存器，寄存器从 \$f0 ~ \$f31。由于这些寄存器位宽为 32 位，因此两个浮点寄存器一起使用可以实现双精度浮点数值。浮点协处理器还有 8 个条件码（cc）标记，序号 0 ~ 7，由比较指令设置，分支（bclf 和 bclt）和条件转移指令完成校验。

lwcl、swcl、mtc1 和 mfcl 指令每次能从寄存器传送或者移出一个字（32 位）。ldcl 和 sdcl 指令，或者像下面描述的 l.s、l.d、s.s 和 s.d 伪指令每次能像寄存器传送或者移出一个双字（64 位）。

在下面的实际指令中，单精度指令的 21 ~ 26 位为 0，双精度指令的 21 ~ 26 位则为 1。在下面的伪指令中，fdest 是浮点寄存器（如 \$f2）。

### 双精度浮点数的绝对值

abs.d fd, fs	0x11	1	0	fs	fd	5
	6	5	5	5	5	6

### 单精度浮点数的绝对值

abs.s fd, fs	0x11	0	0	fs	fd	5
	6	5	5	5	5	6

计算寄存器 fs 中双精度（单精度）浮点数的绝对值，并将计算结果存入寄存器 fd 中。

### 双精度浮点加法

add.d fd, fs, ft	0x11	0x11	ft	fs	fd	0
	6	5	5	5	5	6

A-73

### 单精度浮点加法

add.s	fd, fs, ft	0x11	0x10	ft	fs	fd	0
		6	5	5	5	5	6

计算寄存器 fs 和 ft 中双精度（单精度）浮点数之和，并将计算结果存入寄存器 fd 中。  
浮点数向上舍入

ceil.w.d	fd, fs	0x11	0x11	0	fs	fd	0xe
		6	5	5	5	5	6
ceil.w.s	fd, fs	0x11	0x10	0	fs	fd	0xe

将寄存器 fs 中双精度（单精度）数值向上舍入，并转换成 32 位的定点值，将结果存放  
在寄存器 fd 中。

### 双精度相等比较

c.eq.d	cc, fs, ft	0x11	0x11	ft	fs	cc	0	FC	2
		6	5	5	5	3	2	2	4

### 单精度相等比较

c.eq.s	cc, fs, ft	0x11	0x10	ft	fs	cc	0	FC	2
		6	5	5	5	3	2	2	4

比较寄存器 fs 和 ft 中双精度（单精度）浮点数是否相等，如果相等，将浮点条件标记  
位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

### 双精度小于等于比较

c.le.d	cc, fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xe
		6	5	5	5	3	2	2	4

### 单精度小于等于比较

c.le.s	cc, fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xe
		6	5	5	5	3	2	2	4

将寄存器 fs 和 ft 中双精度（单精度）浮点数进行比较，如果 fs 中的数值小于等于 ft  
中的数值，将浮点条件标记位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

### 双精度小于比较

c.lt.d	cc, fs, ft	0x11	0x11	ft	fs	cc	0	FC	0xc
		6	5	5	5	3	2	2	4

### 单精度小于比较

c.lt.s	cc, fs, ft	0x11	0x10	ft	fs	cc	0	FC	0xc
		6	5	5	5	3	2	2	4

将寄存器 fs 和 ft 中双精度（单精度）浮点数进行比较，如果 fs 中的数值小于 ft 中的  
数值，将浮点条件标记位 cc 设置为 1。如果 cc 被忽略，条件码标记为 0。

### 单精度到双精度的转换

cvt.d.s	fd, fs	0x11	0x10	0	fs	fd	0x21
		6	5	5	5	5	6

### 整型到双精度的转换

cvt.d.w	fd, fs	0x11	0x14	0	fs	fd	0x21
		6	5	5	5	5	6

将寄存器 fs 中的单精度浮点数或者整型数转换成双精度（单精度）浮点数，并存入寄存  
器 fd 中。

### 双精度到单精度的转换

cvt.s.d	fd, fs	0x11	0x11	0	fs	fd	0x20
		6	5	5	5	5	6

### 整型到单精度的转换

cvt.s.w fd, fs	0x11	0x14	0	fs	fd	0x20
	6	5	5	5	5	6

A-75

将寄存器 fs 中的双精度浮点数或者整型数转换成单精度浮点数，并存入寄存器 fd 中。

### 双精度到整型的转换

cvt.w.d fd, fs	0x11	0x11	0	fs	fd	0x24
	6	5	5	5	5	6

### 单精度到整型的转换

cvt.w.s fd, fs	0x11	0x10	0	fs	fd	0x24
	6	5	5	5	5	6

将寄存器 fs 中的双精度浮点数或者单精度浮点数转换成整型数，并存入寄存器 fd 中。

### 双精度浮点除法

div.d fd, fs, ft	0x11	0x11	ft	fs	fd	3
	6	5	5	5	5	6

### 单精度浮点除法

div.s fd, fs, ft	0x11	0x10	ft	fs	fd	3
	6	5	5	5	5	6

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相除，并将计算结果存入寄存器 fd 中。

### 浮点数向下舍入

floor.w.d fd, fs	0x11	0x11	0	fs	fd	0xf
	6	5	5	5	5	6

floor.w.s fd, fs	0x11	0x10	0	fs	fd	0xf
	6	5	5	5	5	6

将寄存器 fs 中的双精度（单精度）数值向下舍入，并将结果存放在寄存器 fd 中。

### 取双精度浮点数

A-76 1.d fdest, address 伪指令

### 取单精度浮点数

1.s fdest, address 伪指令

将地址 address 相应的双精度（单精度）浮点数存入寄存器 fdest 中。

### 双精度浮点数的传送

mov.d fd, fs	0x11	0x11	0	fs	fd	6
	6	5	5	5	5	6

### 单精度浮点数的传送

mov.s fd, fs	0x11	0x10	0	fs	fd	6
	6	5	5	5	5	6

将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

### 条件为假时双精度浮点数传送

movf.d fd, fs, cc	0x11	0x11	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

### 条件为假时单精度浮点数传送

movf.s fd, fs, cc	0x11	0x10	cc	0	fs	fd	0x11
	6	5	3	2	5	5	6

如果条件码标记 cc 为 0，将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。如果 cc 被忽略，条件码标记为 0。

### 条件为真时双精度浮点数传送

movt.d fd, fs, cc	0x11	0x11	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

### 条件为真时单精度浮点数传送

movt.s fd, fs, cc	0x11	0x10	cc	1	fs	fd	0x11
	6	5	3	2	5	5	6

如果条件码标记 cc 为 1，将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。如果 cc 被忽略，条件码标记为 0。

A-77

### 非零条件双精度浮点数传送

movn.d fd, fs, rt	0x11	0x11	rt	fs	fd	0x13
	6	5	5	5	5	6

### 非零条件单精度浮点数传送

movn.s fd, fs, rt	0x11	0x10	rt	fs	fd	0x13
	6	5	5	5	5	6

如果处理器寄存器 rt 中的值不等于 0，那么将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

### 等于零条件双精度浮点数传送

movz.d fd, fs, rt	0x11	0x11	rt	fs	fd	0x12
	6	5	5	5	5	6

### 等于零条件单精度浮点数传送

movz.s fd, fs, rt	0x11	0x10	rt	fs	fd	0x12
	6	5	5	5	5	6

如果处理器寄存器 rt 中的值等于 0，那么将寄存器 fs 中的双精度（单精度）浮点数传送到寄存器 fd 中。

### 双精度浮点乘

mul.d fd, fs, ft	0x11	0x11	ft	fs	fd	2
	6	5	5	5	5	6

### 单精度浮点乘

mul.s fd, fs, ft	0x11	0x10	ft	fs	fd	2
	6	5	5	5	5	6

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相乘，并将计算结果存入寄存器 fd 中。

### 对双精度数求反

neg.d fd, fs	0x11	0x11	0	fs	fd	7
	6	5	5	5	5	6

A-78

### 对单精度数求反

neg.s fd, fs	0x11	0x10	0	fs	fd	7
	6	5	5	5	5	6

对寄存器 fs 中的双精度（单精度）浮点数求反，并将结果存入寄存器 fd 中。

### 对浮点数四舍五入

round.w.d fd, fs	0x11	0x11	0	fs	fd	0xc
	6	5	5	5	5	6

round.w.s fd, fs	0x11	0x10	0	fs	fd	0xc
	6	5	5	5	5	6

将寄存器 fs 中的双精度（单精度）数四舍五入，转换成 32 位的定点数，并存入寄存器 fd 中。

A-79

**对双精度数求平方根**

sqrt.d fd, fs	0x11	0x11	0	fs	fd	4
	6	5	5	5	5	6

**对单精度数求平方根**

sqrt.s fd, fs	0x11	0x10	0	fs	fd	4
	6	5	5	5	5	6

对寄存器 fs 中的双精度（单精度）数求平方根，并存入寄存器 fd 中。

**保存双精度浮点数**

s.d fdest, address 伪指令

**保存单精度浮点数**

s.s fdest, address 伪指令

将寄存器 fdest 中的双精度（单精度）浮点数存入地址 address 中。

**双精度浮点减法**

sub.d fd, fs, ft	0x11	0x11	ft	fs	fd	1
	6	5	5	5	5	6

**A-79 单精度浮点减法**

sub.s fd, fs, ft	0x11	0x10	ft	fs	fd	1
	6	5	5	5	5	6

将寄存器 fs 和 ft 中的双精度（单精度）浮点数相减，并将计算结果存入寄存器 fd 中。

**将浮点数截取为字**

trunc.w.d fd, fs	0x11	0x11	0	fs	fd	0xd
	6	5	5	5	5	6

trunc.w.s fd, fs	0x11	0x10	0	fs	fd	0xd
	6	5	5	5	5	6

对寄存器 fs 中的双精度（单精度）浮点数进行截取操作，转换成 32 位定点数，并将结果存入寄存器 fd 中。

**A. 10.15 异常和中断指令****异常返回**

eret	0x10	1	0	0x18
	6	1	19	6

将协处理器 0 的状态寄存器中的 EXL 位设置为 0，并返回协处理器 0 中 EPC 寄存器指向的指令。

**系统调用**

syscall	0	0	0xc
	6	20	6

寄存器 \$v0 中保存了 SPIM 提供的系统调用的个数（见图 A-9-1）。

**跳出**

break code	0	code	0xd
	6	20	6

产生异常码，异常 1 为调试程序保留。

**空操作**

nop	0	0	0	0	0	0
	6	5	5	5	5	6

A-80 不做任何操作。

## A. 11 小结

用汇编语言进行程序设计需要程序员放弃高级语言中的一些有益的特点——如数据结构、类型检查以及控制结构——以获得对机器执行指令的完全控制。一些应用的外部约束，如响应时间、程序大小等，需要程序员密切关注每条指令。然而，和高级语言程序相比，这种级别的关注带来的是更长、编写更费时、更难维护的汇编语言程序。

此外，三个趋势导致不必再用汇编语言来编写程序。第一个趋势是编译器的改进。现在，编译器生成的代码可以与最好的手工书写的代码相媲美——有时候甚至会更好。第二个趋势是新处理器的速度不仅更快，而且对于那些可以同时执行多条指令的处理器，手工编程也变得更加困难。此外，现代计算机的快速发展也支持高级语言程序不再依赖单一的体系结构。最后，我们见证了日渐复杂的应用趋势，不仅有复杂的图形界面，而且还有许多先前不曾遇见的特征。由程序员组成的团队合作开发的大规模应用程序需要有由高级语言提供的模块化设计思想和语义检查的特点。

## 拓展阅读

Aho, A., R. Sethi, and J. Ullman [1985]. *Compilers: Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley.

*Slightly dated and lacking in coverage of modern architectures, but still the standard reference on compilers.*

Sweetman, D. [1999]. *See MIPS Run*, San Francisco, CA: Morgan Kaufmann Publishers.

*A complete, detailed, and engaging introduction to the MIPS instruction set and assembly language programming on these machines.*

Detailed documentation on the MIPS-32 architecture is available on the Web:

MIPS32™ Architecture for Programmers Volume I: Introduction to the MIPS32™ Architecture  
`(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00082-2B-MIPS32INT-APP-02.00.pdf/getDownload)`

MIPS32™ Architecture for Programmers Volume II: The MIPS32™ Instruction Set  
`(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00086-2B-MIPS32BIS-APP-02.00.pdf/getDownload)`

MIPS32™ Architecture for Programmers Volume III: The MIPS32™ Privileged Resource Architecture  
`(http://mips.com/content/Documentation/MIPSDocumentation/ProcessorArchitecture/ArchitectureProgrammingPublicationsforMIPS32/MD00090-2B-MIPS32PRA-APP-02.00.pdf/getDownload)`

A-81

## A. 12 练习题

- A. 1** [5] < A. 5 > A. 5 节描述了在大多数 MIPS 系统中，内存是如何划分的。请采用其他的方法，实现相同结果。
- A. 2** [20] < A. 6 > 用更少的指令重写 fact 程序。
- A. 3** [5] < A. 7 > 用户程序使用寄存器 \$k0 或 \$k1 时总是安全的么？
- A. 4** [25] < A. 7 > A. 7 节介绍了一种非常简单的异常处理代码。这种处理方式的一个严重的缺陷在于它需要很长的时间来使中断无效。这意味着快速 I/O 设备发出的中断会丢失。请编写更好的可中断的异常处理程序，能尽快使中断有效。
- A. 5** [15] < A. 7 > 简单的异常处理程序总是跳回异常之后的指令。这种操作运行良好，除非导致异常的指令处在分支指令的延迟槽中。这种情况下，下一条指令即是转移的目标。编写更好的程序，

使用 EPC 寄存器来决定异常之后执行哪一条指令。

- A.6** [5] < A.9 > 使用 SPIM，编写、验证一个加法器程序：重复读入整数并对它们相加求和。当输入为 0 时停止程序，并输出累加和。使用 A.9 节介绍的 SPIM 系统调用。
- A.7** [5] < A.9 > 使用 SPIM，编写、验证一个程序：读入三个整数，对两个最大的数求和并输出结果。使用 A.9 节介绍的 SPIM 系统调用。你可以任意中断程序。
- A.8** [5] < A.9 > 使用 SPIM，编写、验证一个程序：使用 SPIM 的系统调用读入一个正整数。如果整数为非正，程序终止，输出“Invalid Entry”；否则程序输出整数每个数字的名称，以空格分隔。例如，如果用户输入“728”，输出“Seven Two Eight”。
- A.9** [25] < A.9 > 用 MIPS 汇编语言编写程序并验证：计算并输出前 100 个素数。如果除了 1 和  $n$  之外没有哪个数能整除  $n$ ，那么  $n$  为素数。你应该实现两个例程：
- `test_prime(n)` 如果  $n$  是素数，返回 1；如果不是，则返回 0。
  - `main()` 循环测试每个整数是否为素数，并输出前 100 个素数。

在 SPIM 上验证你的程序。

- A.10** [10] < A.6, A.9 > 使用 SPIM，编写、验证一个递归程序，来解决汉诺塔问题（需要使用堆栈帧来支持递归）。汉诺塔有三根杆子（1、2 和 3）和  $n$  个盘子（ $n$  是可变的，典型的数值在 1~8 之间）。盘子 1 比盘子 2 小，盘子 2 比盘子 3 小，以此类推，盘子  $n$  是最大的。最开始，所有的盘子都在杆子 1 上，盘子  $n$  在最下面，上面是盘子  $n-1$ ，以此类推，盘子 1 在最上面。目标是将所有的盘子移到杆子 2 上。每次只能移动一个盘子，也就是说，任何一个杆子最上面的盘子只能移到另外两个杆子的顶端。此外，还不能将大盘子放置在小盘子上。

下面的 C 程序会对你用汇编语言编程有所帮助。

```
/* move n smallest disks from start to finish using
extra */

void hanoi(int n, int start, int finish, int extra){
    if(n != 0){
        hanoi(n-1, start, extra, finish);
        print_string("Move disk");
        print_int(n);
        print_string("from peg");
        print_int(start);
        print_string("to peg");
        print_int(finish);
        print_string(".\n");
        hanoi(n-1, extra, finish, start);
    }
}
main(){
    int n;
    print_string("Enter number of disks>");
    n = read_int();
    hanoi(n, 1, 2, 3);
    return 0;
}
```

A-83