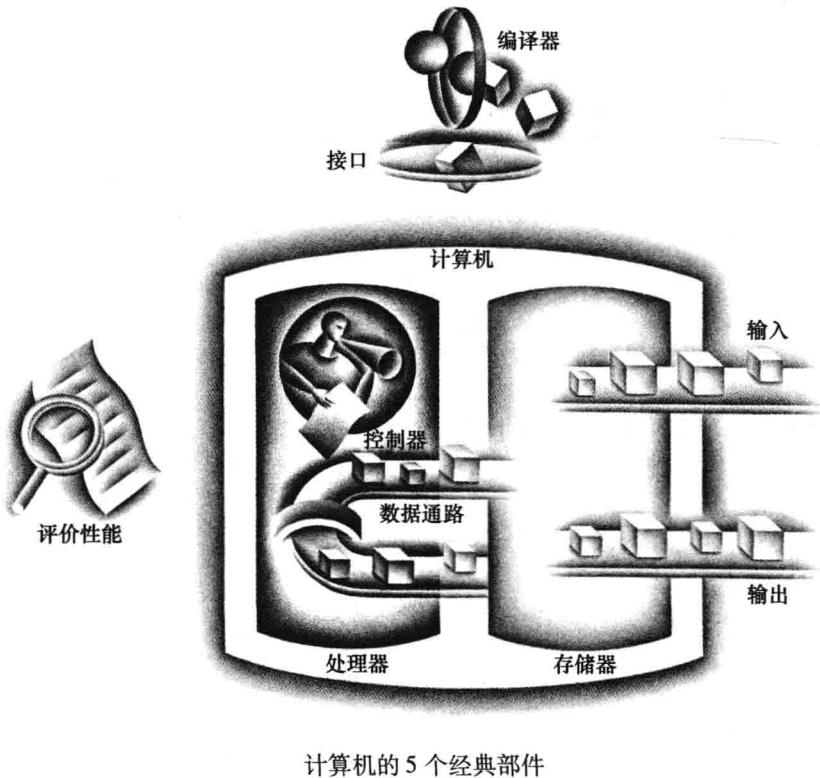


第2章 |

Computer Organization and Design: The Hardware/Software Interface

指令：计算机的语言



我对上帝说西班牙语，对女人说意大利语，对男人说法语，对我的马说德语。

——法王查理五世 (1500—1558)

2.1 引言

要计算机服从指挥，就必须用计算机的语言。计算机语言中的基本单词称为指令，一台计算机的全部指令称为该计算机的指令集（instruction set）。本章将介绍实际计算机指令集的两种形式：一种是人们编程书写的“汇编语言助记符”，另一种是计算机所能识别的形式。我们将以自顶向下的方式来介绍，从看似受约束的汇编语言助记符开始，逐步精炼到实际计算机的真实语言。第3章将继续采用这种向下探究的方式，揭示算术运算的硬件以及浮点数的表示方法。

● 指令集：一个给定的计算机体系结构所包含的指令集合。

尽管机器语言种类繁多，但它们之间十分类似，其差异性更像人类语言中的“方言”，而非各自独立的语言。因此，理解了一种机器语言，其他种类的机器语言也就容易理解了。

本书选择 MIPS 技术的指令集，它是自 20 世纪 80 年代以来出现的优秀指令集。通过简要介绍其他三种流行的指令集可以看出 MIPS 指令的优势。

1) ARMv7 与 MIPS 类似。2011 年，ARM 处理器芯片的产量超过 90 亿片，这使得 ARMv7 成为最流行的指令集。

- 2) 第二个例子是 Intel x86，在 PC 领域和后 PC 时代的云计算领域占统治地位。
- 3) 第三个例子是 ARMv8，它将 ARMv7 的地址范围由 32 位扩展到 64 位。而具有讽刺意味的是，这个 2013 年产生的指令集更加接近于 MIPS，而非 ARMv7。

这种相似性一方面是因为所有计算机都是基于基本原理相似的硬件技术所构建的，另一方面是因为所有计算机都必须提供一些基本操作。此外，计算机设计者有一个共同的目标：找到一种语言，可方便硬件和编译器的设计，且使性能最佳，同时使成本和功耗最低。但实现这个目标需要长期的探索。下述引文写于计算机出现不久的 1947 年，但今天它仍然适用：

用形式逻辑的方法可以很容易看到，在理论上存在着某种“指令集”足以控制任何的操作序列并使之执行……从当前的观点出发，在选择一个“指令集”时，真正的决定性因素是要更多地考虑其实际性质：“指令集”要求的设备简单性，它的应用对于解决实际重要问题的明确性以及它解决该类问题的处理速度。

——Burks, Goldstine, von Neumann, 1947

无论是对 20 世纪 50 年代的计算机而言，还是对现代的计算机来说，“设备简单性”都是值得考虑的重要问题。本章的目的就是讲解符合此原则的一种指令集，介绍它怎样用硬件表示，以及它和高级编程语言之间的关系。我们的示例使用 C 语言编写，2.15 节介绍了在使用像 Java 这样的面向对象语言时会有什么不同。

通过理解如何表述指令，读者也将发现计算的秘密：存储程序概念（stored-program concept）。此外，通过使用机器语言编程，并在本书提供的模拟器中运行，读者将进一步体会到编程语言和编译优化对程序性能的影响。本章结束时我们将简要介绍指令集的发展历史和其他的计算机“方言”。

◎ 存储程序概念：多种类型的指令和数据均以数字形式存储于存储器中的概念，存储程序型计算机即源于此。

我们结合计算机的结构，逐步讲解 MIPS 指令集。采用自顶向下、循序渐进的方法并结合各部件及其说明，尽量使机器语言变得不再枯燥。图 2-1 给出了本章将要介绍的指令集的总体情况。

MIPS 操作数

名字	示例	注释
32 个寄存器	\$s0 - \$s7, \$t0 - \$t9, \$zero, \$a0 - \$a3, \$v0 - \$v1, \$gp, \$fp, \$sp, \$ra, \$at	寄存器用于数据的快速存取。在 MIPS 中，只能对存放在寄存器中的数据执行算术操作，寄存器 \$zero 的值恒为 0，寄存器 \$at 被汇编器保留，用于处理大的常数
2 ³⁰ 个存储器字	Memory [0], Memory [4], …, Memory [4294967292]	存储器只能通过数据传输指令访问。MIPS 使用字节编址，所以连续的字地址相差 4。存储器用于保存数据结构、数组和溢出的寄存器

MIPS 汇编语言

类别	指令	示例	含义	注释
算术	加法	add \$s1, \$s2, \$s3	$s1 = s2 + s3$	三个寄存器操作数
	减法	sub \$s1, \$s2, \$s3	$s1 = s2 - s3$	三个寄存器操作数
	立即数加法	addi \$s1, \$s2, 20	$s1 = s2 + 20$	用于加常数数据

图 2-1 本章要讲解的是 MIPS 汇编语言指令。示例含义注释信息也可以在 MIPS 参考数据卡的第 1 列中找到

MIPS 汇编语言

类别	指令	示例	含义	注释
数据传输	取字	lw \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字从内存中取到寄存器中
	存字	sw \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字从寄存器中取到内存中
	取半字	lh \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将半个字从内存中取到寄存器中
	取无符号半字	lhu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将半个字从内存中取到寄存器中
	存半字	sh \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将半个字从寄存器存到内存中
	取字节	lb \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字节从内存取到寄存器中
	取无符号字节	lbu \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	将一个字节从内存取到寄存器中
	存字节	sb \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1	将一个字节从寄存器存到内存中
	取链接字	ll \$s1,20(\$s2)	\$s1 = Memory[\$s2 + 20]	取字作为原子交换的前半部
	存条件字	sc \$s1,20(\$s2)	Memory[\$s2 + 20] = \$s1; \$s1 = 0 or 1	存字作为原子交换的后半部分
逻辑	取立即数的高位	lui \$s1,20	\$s1 = 20 * 2 ¹⁶	取立即数并放到高16位
	与	and \$s1,\$s2,\$s3	\$s1 = \$s2 & \$s3	三个寄存器操作数按位与
	或	or \$s1,\$s2,\$s3	\$s1 = \$s2 \$s3	三个寄存器操作数按位或
	或非	nor \$s1,\$s2,\$s3	\$s1 = ~(\$s2 \$s3)	三个寄存器操作数按位或非
	立即数与	andi \$s1,\$s2,20	\$s1 = \$s2 & 20	和常数按位与
	立即数或	ori \$s1,\$s2,20	\$s1 = \$s2 20	和常数按位或
	逻辑左移	sll \$s1,\$s2,10	\$s1 = \$s2 << 10	根据常数左移相应位
条件分支	逻辑右移	srl \$s1,\$s2,10	\$s1 = \$s2 >> 10	根据常数右移相应位
	相等时跳转	beq \$s1,\$s2,25	if(\$s1 == \$s2) go to PC + 4 + 100	相等检测； 和PC相关的跳转
	不相等时跳转	bne \$s1,\$s2,25	if(\$s1 != \$s2) go to PC + 4 + 100	不相等检测； 和PC相关的跳转
	小于时置位	slt \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	比较是否小于； beq, bne
	无符号数比较小于时置位	sltu \$s1,\$s2,\$s3	if(\$s2 < \$s3) \$s1 = 1; else \$s1 = 0	比较是否小于无符号数
	无符号数比较小于立即数时置位	slti \$s1,\$s2,20	if(\$s2 < 20) \$s1 = 1; else \$s1 = 0	比较是否小于常数
无条件跳转	无符号数比较小于无符号立即数时置位	sltiu \$s1,\$s2,20	if(\$s2 < 20) \$s1 = 1; else \$s1 = 0	比较是否小于无符号常数
	跳转	j 2500	go to 10000	跳转到目标地址
	跳转至寄存器所指位置	jr \$ra	go to \$ra	用于switch语句， 以及过程调用
	跳转并链接	jal 2500	\$ra = PC + 4; go to 10000	用于过程调用

图 2-1 (续)

2.2 计算机硬件的操作

毫无疑问，计算机必须有执行基本算术运算操作的指令。

——Burks、Goldstine、von Neumann, 1947

任何计算机必须能够执行算术运算。MIPS 汇编语言的下述记法

```
add a, b, c
```

表示将两个变量 b 和 c 相加，并将它们的和放入变量 a 中。

这种记法的表示方式是固定的：每条 MIPS 算术指令只执行一个操作，并且有且仅有 3 个变量。例如，若要将变量 b、c、d、e 之和放入变量 a 中（本节不深究“变量”的含义，下一节将给出其详细说明）。

下面的指令序列将完成此 4 个变量的相加：

```
add a, b, c    # The sum of b and c is placed in a
add a, a, d    # The sum of b, c, and d is now in a
add a, a, e    # The sum of b, c, d, and e is now in a
```

以上 3 条指令完成了 4 个变量的相加。

上述每行代码中，符号“#”右边的是注释，用于帮助人们理解程序，而计算机将忽略它们。注意与其他编程语言不同的是，这种语言的每一行最多只有一条指令。另一个与 C 语言不同的是，注释总是在一行的末尾结束。

与加法类似的指令一般都有三个操作数：两个进行运算的数和一个保存结果的数。要求每条指令有且仅有三个操作数，这一点符合硬件简单性的设计原则：操作数个数可变将给硬件设计带来更大的复杂性。这种情况说明了硬件设计三条基本原则的第一条：

设计原则 1：简单源于规整。

下面的两个示例程序展示了用高级编程语言编写的程序和用汇编语言编写的程序之间的关系。

01 例题·把 C 语言中两条赋值语句编译成 MIPS

本例中 C 语言程序包含 5 个变量，a、b、c、d 和 e。因为 Java 语言由 C 语言演化而来，所以本例及以后若干例子对这两种高级语言均适用：

```
a = b + c;
d = a - e;
```

将 C 语言程序转换为 MIPS 汇编指令是由编译器完成的。写出由编译器生成的 MIPS 代码。

01 答案

一条 MIPS 指令对来自两个源操作数寄存器的操作数进行运算，并将结果存入目的寄存器。因此上面两条简单的 C 语句可直接编译为如下两条 MIPS 汇编指令：

```
add a, b, c
sub d, a, e
```

01 例题·把 C 语言中一条复杂的赋值语句编译成 MIPS

下面一行复杂的 C 语句包含 5 个变量 f、g、h、i 和 j：

```
f = (g + h) - (i + j);
```

C 编译器将产生什么样的 MIPS 汇编语言代码？

01 答案

因为一条 MIPS 指令仅执行一个操作，所以编译器必须将这条 C 语句编译成多条汇编指令。若第一条指令计算 g 与 h 的和，其结果必须暂存在某一个地方。因此，编译器需创建一个临时

变量 t0:

```
add t0,g,h # temporary variable t0 contains g + h
```

虽然下一个操作是减法，但在做减法操作之前，必须先计算出 i 与 j 的和。因此，第二条指令将 i、j 之和存于由编译器创建的另一个临时变量 t1 中：

```
add t1,i,j # temporary variable t1 contains i + j
```

最后，用一条减法指令将两个临时变量中的值相减，结果存入变量 f，完成编译：

```
sub f,t0,t1 # f gets t0 - t1, which is (g + h) - (i + j)
```

□

01 小测验

对于一个给定的功能，用下列哪种编程语言实现可能花费的代码行数最多？将下面 3 种语言排序：

1. Java
2. C
3. MIPS 汇编语言

01 精解 为了增强可移植性，Java 最初被设定为依靠软件解释器执行的语言。解释器的指令集称作 Java 字节码（Java bytecode，参见 2.15 节），它与 MIPS 指令集有很大不同。为使性能与等效功能的 C 程序接近，Java 系统现在的典型做法是将字节码编译成类似 MIPS 这样的机器指令。因为通常 Java 完成编译的时间迟于 C，所以 Java 编译器常称为即时编译器（Just In Time，JIT）。2.12 节展示了在程序启动阶段 JIT 是如何迟于 C 编译器的，2.13 节展示了 Java 程序的编译执行和解释执行的性能比较。

2.3 计算机硬件的操作数

与高级语言程序不同，MIPS 算术运算指令的操作数是很严格的，它们必须来自寄存器。寄存器由硬件直接构建且数量有限，是计算机硬件设计的基本元素。当计算机设计完成后，寄存器对程序员是可见的，所以也可以把寄存器想象成构造计算机“建筑”的“砖块”。在 MIPS 体系结构中寄存器大小为 32 位，由于 32 位为一组的情况经常出现，因此在 MIPS 体系结构中将其称为字（word）。

66 ◀ 字：计算机中的基本访问单位，通常是 32 位为一组，在 MIPS 体系结构中与寄存器大小相同。

高级语言的变量与寄存器的一个主要区别在于寄存器的数量有限，MIPS 一类的典型的现代计算机中有 32 个寄存器（参见 2.21 节有关寄存器数目的演变历史）。下面继续以自顶向下的方式引入新的 MIPS 语言的符号表示。在本节中 MIPS 算术指令的三个操作数限定为必须从 32 个 32 位寄存器中选取。

寄存器个数限制为 32 个的理由可以表示为硬件设计三条基本原则中的第二条：

设计原则 2：越小越快。

大量的寄存器可能会使时钟周期变长，因为电信号传输更远的距离必然花费更长的时间。

当然，该原则也不是绝对的，31 个寄存器不见得比 32 个更快。但表象背后的物理事实值得计算机设计者认真对待。在这种情况下，设计者必须在程序期望更多寄存器和加快时钟周期之间进行权衡。另一个不使用多于 32 个寄存器的原因是受指令格式位数的限制，这在 2.5 节有相应介绍。

第4章论证了寄存器在硬件结构中所扮演的核心角色。正如该章所述，有效利用寄存器对于提高程序性能极为重要。

尽管可以简单使用序号0~31表示相应的寄存器，但MIPS约定书写指令时用一个“\$”符后面跟两个字符来代表一个寄存器。2.8节将解释这一做法的理由。现在，我们使用\$s0, \$s1, …来表示与C和Java程序中的变量所对应的寄存器；用\$t0, \$t1, …来表示将程序编译为MIPS指令时所需的临时寄存器。

01 例题·使用寄存器编译C赋值语句

将程序变量和寄存器对应起来是编译器的工作。以我们前面讲过的C赋值语句为例：

$f = (g + h) - (i + j);$

变量f、g、h、i和j依次分配给寄存器\$s0、\$s1、\$s2、\$s3和\$s4。编译后的MIPS代码是什么？

67

01 答案

除了将变量用上述寄存器代替，将两个临时变量用\$t0和\$t1代替外，编译后生成的代码与前面例题中的代码非常相似：

```
add $t0,$s1,$s2 # register $t0 contains g + h
add $t1,$s3,$s4 # register $t1 contains i + j
sub $s0,$t0,$t1 # f gets $t0 - $t1, which is (g + h)-(i + j)
```

□

2.3.1 存储器操作数

编程语言中，有像上面这些例题中仅含一个数据元素的简单变量，也有像数组或结构那样的复杂数据结构。这些复杂数据结构中的数据元素可能远多于计算机中寄存器的个数。计算机怎样来表示和访问这样大的结构呢？

回忆一下第1章所描述的计算机的5个组成部分。处理器只能将少量数据保存在寄存器中，但存储器有数十亿的数据元素。因此，数据结构（如数组和结构）是存放在存储器中的。

如上所述，MIPS的算术运算指令只对寄存器进行操作，因此，MIPS必须包含在存储器和寄存器之间传送数据的指令。这些指令叫作**数据传送指令**（data transfer instruction）。为了访问存储器中的一个字，指令必须给出存储器地址（address）。存储器就是一个很大的下标从0开始的一维数组，地址就相当于数组的下标。例如，在图2-2中，第三个数据元素的地址为2，存放的数据为10。

- ② 数据传送指令：在存储器和寄存器之间移动数据的命令。
- ② 地址：用于在存储器空间中指明某特定数据元素位置的值。

将数据从存储器复制到寄存器的数据传送指令通常叫取数（load）指令。取数指令的格式是操作码后接着目标寄存器，再后面是用来访问存储器的常数和寄存器。常数和第二个寄存器中的值相加即得存储器地址。实际的MIPS取数指令助记符为lw，为load word的缩写。

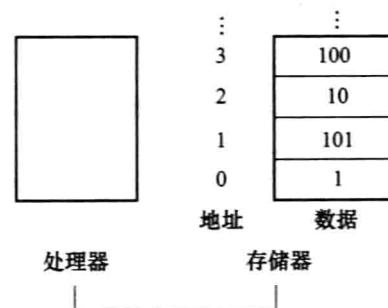


图2-2 存储器地址和该地址对应的数据。如果这些元素是字，那么这些地址就是错误的，因为MIPS实际上是按字节编址的，而一个字是4字节。图2-3给出了顺序字编址的内存寻址

68

01 例题·编译一个操作数在存储器中的 C 赋值语句

设 A 是一个含有 100 个字的数组，像前面的例题一样，编译器仍然将寄存器 \$s1、\$s2 依次分配给变量 g、h。又设数组 A 的起始地址（或称基址（base address））存放在寄存器 \$s3 中。试编译下面的 C 赋值语句：

```
g = h + A[8];
```

01 答案

虽然该 C 赋值语句只有一个操作，但其中一个操作数在存储器中，所以首先必须将 A [8] 传递到寄存器中。其地址是 \$s3 中的基址加上该元素序号 8。取回的数据应放在一个临时寄存器中以便下条指令使用。由图 2-2 可知，编译后生成的第一条指令为：

```
lw      $t0,8($s3) # Temporary reg $t0 gets A[8]
```

（这里是一种简化版描述，后面会对这条指令做相关的微调。）因为 A [8] 已取到寄存器 \$t0 中，下一条指令就可对 \$t0 进行操作。该指令将 h（在 \$s2 中）加上 A [8]（在 \$t0 中），并将结果放到对应于 g 的寄存器 \$s1 中：

```
add    $s1,$s2,$t0 # g = h + A[8]
```

数据传送指令中的常量（本例中为 8）称为偏移量（offset），存放基址的寄存器（本例中为 \$s3）称为基址寄存器（base register）。□

01 硬件/软件接口 除了将变量与寄存器对应起来，编译器还在存储器中为诸如数组和结构这样的数据结构分配相应的位置。然后，编译器可以将它们在存储器中的起始地址放到数据传送指令中。

很多程序都用到 8 比特的字节类型，且大多数体系结构按字节编址。因此，一个字的地址必和它所包括的 4 字节中某个的地址相匹配，且连续字的地址相差 4。例如，图 2-3 给出了图 2-2 的实际 MIPS 地址，其中第三个字的字节地址是 8。

因为 MIPS 是按字节编址的，所以字的起始地址必须是 4 的倍数。这叫对齐限制（alignment restriction），许多体系结构都有这样的限制（第 4 章说明了对齐能加快数据传送的理由）。

有两种类型的字节寻址的计算机：一种使用最左边或“大端”（big end）字节的地址作为字地址；另一种使用最右边或“小端”（little end）字节的地址作为字地址。MIPS 采用的是大端编址（big-endian）。由于使用相同的地址去访问一个字和 4 个字节时“端”才起作用，因此大多数情况下不需要关注该问题。（附录 A 中给出了在一个字中对字节进行记数的两种方法。）

字节寻址也影响到数组下标。在上面的代码中，为了得到正确的字节地址，与基址寄存器 \$s3 相加的偏移量必须是 4×8 ，即 32，这样才能正确读到 A [8]，而不会错读到 A [8/4]。（参见 2.19 节中相关陷阱的介绍。）

与取数指令相对应的指令通常叫作存数（store）指令；它将数据从寄存器复制到存储器。存数指令的格式和取数指令相似：首先是操作码，接着是包含待存储数据的寄存

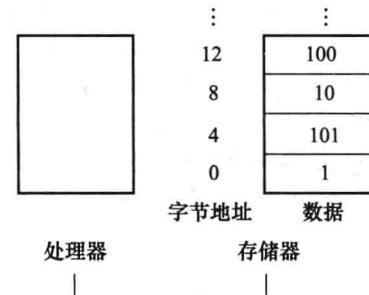


图 2-3 实际的 MIPS 存储器地址和该地址对应的数据。相对于图 2-2，变化的地址采用高亮显示。由于 MIPS 按字节编址，而字地址是 4 的倍数，因为每个字的长度为 4 字节

器，然后是数组元素的偏移量，最后是基址寄存器。同样，MIPS 地址由常数和基址寄存器内容共同决定。实际的 MIPS 存数指令为 sw，即 store word 的缩写。

- 对齐限制：数据地址与存储器的自然边界对齐的要求。

69

01 硬件/软件接口 由于 load 和 store 指令中的地址是二进制，因此作为主存的 DRAM 的容量使用二进制表示，而非十进制。例如，使用 gibibytes (2^{30}) 或 tebibytes (2^{40}) 表示，而不用 gigabytes (10^9) 或 terabytes (10^{12})，见图 1-1。

70

01 例题·用取数/存数指令进行编译

假设变量 h 存放在寄存器 \$s2 中，数组 A 的基址放在 \$s3 中。试编译下面的 C 赋值语句：
 $A[12] = h + A[8];$

01 答案

虽然该 C 语句只有一个操作，但是有两个操作数在存储器中，因此，需要更多的 MIPS 指令。前两条指令基本上与上个例题相同，除了本例在取数指令中选择 A [8] 时使用了字节寻址中正确的偏移量，并且加法指令将结果放在临时寄存器 \$t0 中：

```
lw    $t0,32($s3)  # Temporary reg $t0 gets A[8]
add  $t0,$s2,$t0  # Temporary reg $t0 gets h + A[8]
```

最后一条指令使用 48 (4×12) 作为偏移量，寄存器 \$s3 作为基址寄存器，将加法结果存放到存储器单元 A [12] 中。

```
sw    $t0,48($s3)  # Stores h + A[8] back into A[12]
```

lw 和 sw 是 MIPS 体系结构中在存储器和寄存器之间复制字的指令。其他计算机有各自相应的取数/存数指令来传送数据。Intel x86 体系结构中类似的指令见 2.17 节。 □

01 硬件/软件接口 许多程序的变量个数要远多于计算机的寄存器个数。因此，编译器会尽量将最常用的变量保持在寄存器中，而将其他的变量放在存储器中，方法是使用取数/存数指令在寄存器和存储器之间传送变量。将不常使用的变量（或稍后才使用的变量）存回到存储器中的过程叫作寄存器溢出（spilling）。

根据硬件设计原则 2，存储器一定比寄存器慢，因为寄存器数量更少。事实的确如此，访问寄存器中的数据要远快于访问存储器中的数据。

另外，寄存器中的数据更容易利用。一条 MIPS 算术运算指令能完成读两个寄存器、对它们进行运算以及写回运算结果的操作。而一条 MIPS 数据传送指令只能完成读一个操作数或写一个操作数的操作，并且不能对它们进行运算。

寄存器与存储器相比，访问时间短、吞吐率高，寄存器中的数据访问速度快并易于利用，访问寄存器相对于访问存储器功耗更小。因此，为了获得高性能和节约功耗，指令集的体系结构必须拥有足够的寄存器，并且编译器必须高效率地利用这些寄存器。

2.3.2 常数或立即数操作数

程序中经常会在某个操作中使用到常数——例如，将数组的下标加 1，用以指向下一个数组元素。实际上，在运行 SPEC CPU 2006 测试基准程序集时，有超过一半的 MIPS 算术运算指令会用到常数作为操作数。

仅从已介绍过的指令看，如果要使用常数必须先将其从存储器中取出。（常数可能是在程序被加载时放入存储器的。）例如，要使寄存器 \$s3 加 4，可以使用下面的代码：

71

```
lw $t0, AddrConstant4($s1)    # $t0 = constant 4
add $s3,$s3,$t0                # $s3 = $s3 + $t0 ($t0 == 4)
```

假设 $\$s1 + \text{AddrConstant4}$ 是常量 4 的存储器地址。

避免使用取数指令的另一方法是，提供其中一个操作数是常数的算术运算指令。这种有一个常数操作数的快速加法指令叫作加立即数 (add immediate)，或者写成 addi。这样，上述操作只需写成：

```
addi    $s3,$s3,4           # $s3 = $s3 + 4
```

常数操作数出现频率高，而且相对于从存储器中取常数，包含常数的算术运算指令执行速度快很多，并且能耗较低。

常数 0 还有另外的作用，有效使用它可以简化指令集。例如，数据传送指令正好可以被视作一个操作数为 0 的加法。因此，MIPS 将寄存器 $\$zero$ 恒置为 0。(此寄存器编号也为 0。)根据使用频率来确定要定义的常数是加速大概率事件的另一个好办法。

01 小测验

根据寄存器的重要性，芯片中寄存器数目随时间的增长率符合下面哪种情况？

1. 非常快：像摩尔定律一样快，该定律预测，芯片上的晶体管数目每 18 个月翻一番。
2. 非常慢：由于程序是通过计算机语言实现的，而指令集体系结构具有惯性，因此寄存器数目的增长要与新指令集的可行性保持一致。

72 **01 精解** 虽然本书中讲到的 MIPS 寄存器都是 32 位的，但是也有 64 位版本的 MIPS 指令集，它具有 32 个 64 位的寄存器。为了加以区分，分别将它们称为 MIPS-32 和 MIPS-64。在本章中，我们使用 MIPS-32 的子集。附录 E 中介绍了 MIPS-32 和 MIPS-64 的区别。2.16 节和 2.18 节介绍了 ARMv7 的 32 位地址和 ARMv8 的 64 位地址之间更多显著的差别。

01 精解 MIPS 中偏移量加基址寄存器的寻址方式非常适合数组和结构，因为基址寄存器可指向结构的首地址，偏移量可用于选择所需的数据元素。在 2.13 节中我们将看到这样的例子。

01 精解 最初设计数据传送指令时，基址寄存器用于保存数组下标，而偏移量用来标示数组的起始地址。因而基址寄存器也叫作下标寄存器 (index register)。现在，存储器容量大大增加，数据分配的软件模型也更为复杂，所以数组的基地址通常放在寄存器中。如同下面将要看到的那样，基地址可能由于过大而不适宜用偏移量表示。

01 精解 由于 MIPS 支持负常数，所以 MIPS 中不需要设置减立即数的指令。

2.4 有符号数和无符号数

首先让我们快速回顾一下计算机是如何表示数的。我们所受的教育是以十进制为基础的，但数的进制可以是任意的。例如，十进制的 123 等于二进制的 1111011。

在计算机硬件中，数是以一串或高或低的电信号来体现的，这恰好可以被认为是基为 2 的数（与基为 10 的数称为十进制数一样，基为 2 的数称为二进制数）。

所有信息都由二进制数位 (binary digit) 或位 (bit) 组成，因此二进制数运算基本单位是 bit，取值可以是两种状态之一：高或低，开或关，真或假，1 或 0。

② 二进制数位：也称二进制位，二进制状态之一，即 0 或 1，是信息的基本组成单位。

推广到任意进制，第 i 位 d 的值是

$$d \times \text{Base}^i$$

这里， i 是从 0 开始并且从右向左递增。显而易见，计算一个数各位数值的方法是使用幂。我们在十进制数的右下角写上 10，在二进制数的右下角写上 2。例如，

1011_2

表示

$$\begin{aligned} & (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)_{10} \\ & = (1 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)_{10} \\ & = 8 + 0 + 2 + 1_{10} \\ & = 11_{10} \end{aligned}$$

73

在一个 32 位的字中，我们从右向左标记各位为 0, 1, 2, 3…，下面的图片表示了 MIPS 字中每一位的编号和数字 1011_2 的存放位置。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	

(32位宽)

由于字是在水平或垂直方向上书写的，用最左边或最右边表示大小带有不确定性，因此采用最低有效位 (least significant bit) 表示最右边的一位（上图中的第 0 位），最高有效位 (most significant bit) 表示最左边的一位（上图中的第 31 位）。

- ② 最低有效位：在 MIPS 字中最右边的一位。
- ② 最高有效位：在 MIPS 字中最左边的一位。

MIPS 的字有 32 位，可以表示 2^{32} 个不同的 32 位模式。很自然就可以使这些组合表示从 0 到 $2^{32} - 1$ ($4\,294\,967\,295_{10}$) 之间的数：

$$\begin{aligned} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &= 0_{10} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &= 1_{10} \\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &= 2_{10} \\ \dots \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 &= 4,294,967,293_{10} \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &= 4,294,967,294_{10} \\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &= 4,294,967,295_{10} \end{aligned}$$

如下式，32 位的二进制数字也可以表示成每位的值乘以该位对应的 2 的幂次的形式（这里 x_i 表示数字 x 的第 i 位）：

$$(x31 \times 2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \dots + (x1 \times 2^1) + (x0 \times 2^0)$$

在稍后我们将看到，由于一些原因，这些正数被称为无符号数。

01 硬件/软件接口 二进制对人类来说不是自然的计数方法，我们有 10 个手指头，所以我们自然会采用十进制数。为什么计算机不使用十进制呢？事实上，第一台商用计算机确实提供了十进制算术。问题在于计算机仍然采用开关信号，所以一个十进制数将由几个二进制数来表示。事实证明十进制效率很低，所以后来的计算机都转向了二进制，只有在相对很少发生的 I/O 事件中才将数据转换成十进制。

需要注意的是，上式是二进制数的一般表示。实际上，数是由无穷多的位组成的，其中除了最右边的少数位以外其余大部分都是 0。正常情况下不用表示左边的 0。

硬件可以对这些二进制数进行加、减、乘、除操作。如果操作结果不能被最右端的硬件位所表示，那么就发生了溢出 (overflow)。如何处理溢出是由编程语言、操作系统和程序来决定的。

74

计算机程序对正数和负数都要进行计算，所以需要一种方法来区分正数和负数。显而易见的解决方案是增加一个独立的符号位，这种表示方法称为符号和幅值（sign and magnitude）表示法。

符号和幅值表示法有若干缺点。首先，符号位放在哪里不够明确，放在右边还是左边？早期的计算机对两种方法都尝试过。其次，因为不可能在计算时提前得知结果的符号，对于符号和幅值表示的数进行计算需要额外的一步来设置符号。最后，一个单独的符号位意味着在符号和幅值表示的数中不但有正零而且还有负零，这将给粗心的程序员带来问题。这些缺点导致这种表示方法很快就被放弃了。

在研究更具吸引力的替代方案时产生了这样一个问题，当我们试图用一个较小的数减去一个较大的数时，无符号数表示方法的结果将会是什么？答案是较小的数字将会从前面的0中借位，所有结果中前面的位都变成了一串1。

在没有其他明显更好选择的情况下，最终的解决方案是选择一种易于硬件实现的表达方式：前导位为0表示正数，前导位为1表示负数。这种常用的表示有符号二进制数的方法称为二进制补码（two's complement）。例如：

0000 0000 0000 0000 0000 0000 0000 0000 ₂	= 0 ₁₀
0000 0000 0000 0000 0000 0000 0000 0001 ₂	= 1 ₁₀
0000 0000 0000 0000 0000 0000 0000 0010 ₂	= 2 ₁₀
...	...
0111 1111 1111 1111 1111 1111 1111 1101 ₂	= 2,147,483,645 ₁₀
0111 1111 1111 1111 1111 1111 1111 1110 ₂	= 2,147,483,646 ₁₀
0111 1111 1111 1111 1111 1111 1111 1111 ₂	= 2,147,483,647 ₁₀
1000 0000 0000 0000 0000 0000 0000 0000 ₂	= -2,147,483,648 ₁₀
1000 0000 0000 0000 0000 0000 0000 0001 ₂	= -2,147,483,647 ₁₀
1000 0000 0000 0000 0000 0000 0000 0010 ₂	= -2,147,483,646 ₁₀
...	...
1111 1111 1111 1111 1111 1111 1111 1101 ₂	= -3 ₁₀
1111 1111 1111 1111 1111 1111 1111 1110 ₂	= -2 ₁₀
1111 1111 1111 1111 1111 1111 1111 1111 ₂	= -1 ₁₀

上面的数字中一半是正数，从 $0 \sim 2,147,483,647_{10}$ ($2^{31} - 1$)，这些数字的表示方式与之前是一样的。紧接着的 $1000\cdots0000_2$ 表示最小的负数 $-2,147,483,648_{10}$ (-2^{31})。而后是按照绝对值递减的负数：从 $-2,147,483,647_{10}$ ($1000\cdots0001_2$) 到 -1_{10} ($1111\cdots1111_2$)。

二进制补码中的最小负数 $-2,147,483,648_{10}$ 没有相应的正数与之对应。这种不平衡同样也会为粗心的程序员带来烦恼，但相比符号和幅值方法，该方法不会对程序员和硬件设计人员造成困扰。因此，现在所有计算机都采用二进制补码方法来表示有符号数。

采用二进制补码方法的优点在于所有负数的最高有效位都是1。硬件只需检测这一位就可以知道一个数是正数还是负数（这一位为0表示是正数）。因此，这个位通常叫作符号位。在理解了符号位之后，就可以使用2的幂次的方式来表示正的和负的32位数：

$$(x31 \times -2^{31}) + (x30 \times 2^{30}) + (x29 \times 2^{29}) + \cdots + (x1 \times 2^1) + (x0 \times 2^0)$$

符号位被 -2^{31} 乘，其余的位仍按前面的方法计算。

01 例题·二进制到十进制的转换

下面这个用32位二进制补码表示的数对应的十进制数是多少？

1111 1111 1111 1111 1111 1111 1111 1100₂

01 答案

将数的位值代入上面的公式：

$$(1 \times (-2^{31})) + (1 \times 2^{30}) + (1 \times 2^{29}) + \cdots + (1 \times 2^2) + (0 \times 2^1) + (0 \times 2^0) \\ = -2^{31} + 2^{30} + 2^{29} + \cdots + 2^2 + 0 + 0$$

$$\begin{aligned}
 &= -2\ 147\ 483\ 648_{10} + 2\ 147\ 483\ 644_{10} \\
 &= -4_{10}^{\ominus}
 \end{aligned}$$

后面将给出从负数转换为正数的捷径。 □

就像无符号数的操作结果可能超过硬件允许的容量而发生溢出一样，对二进制补码数的操作也可能发生溢出。溢出发生在有限二进制数最左边的符号位与采用无穷多位表示该数时左边位的值不同的情况下（即符号位不正确）：该数是负数时符号位是 0，或该数是正数时符号位是 1。

01 硬件/软件接口 和算术运算一样，对取数指令来说有符号数和无符号数是有区别的。取回有符号数后需要使用符号位填充寄存器的所有剩余位，称为符号扩展，但其目的是在寄存器中放入数字正确的表示方式。取回无符号数只是简单地用 0 来填充数据左侧的剩余位，因为这种表示形式的数是没有符号的。

当把 32 位的字加载到 32 位的寄存器中时，上面的讨论是没有意义的，因为无符号数和有符号数的加载是完全一样的。MIPS 提供了两种字节加载的方法：一种是用于字节加载的 `lb` (`load byte`)，`lb` 将字节看作有符号数，使用符号扩展来填充寄存器的左侧 24 位；另一种是用于无符号整数加载的 `lbu` (`load byte unsigned`)。由于 C 程序几乎都是使用字节来表示字符，很少用来表示有符号短整数 (`short signed integers`)，所以实际中几乎所有字节加载都使用 `lbu`。

01 硬件/软件接口 与上面所讨论的数不同，存储器地址很自然地从 0 开始一直连续增加到最大的地址。换言之，负地址是没有意义的。因此，程序有时需要处理一些可以是正也可以是负的数，有时需要处理一些仅能是正的数。一些编程语言反映了这个区别。例如，C 语言将前者叫作整数 (`int`) 而后者叫作无符号整数 (`unsigned int`)。一些 C 编程风格的指导书甚至推荐用 `signed int` 来声明前一种数，以使区别更加明显。

我们来看两种处理二进制补码数的简单方法。第一种是对二进制补码数取反的快速方法。简单对每一位取反，0 变成 1，1 变成 0，然后对结果加 1。这种方法是基于这样的事实，一个数和它按位取反的结果相加，和一定是 $111\dots111_2$ ，即 -1 。因此 $x + \bar{x} = -1$ ，即 $x + \bar{x} + 1 = 0$ 或 $\bar{x} + 1 = -x$ 。（我们使用 \bar{x} 表示将 x 的每位取反）。

01 例题·求反的捷径

对 2_{10} 求反，然后通过对 -2_{10} 求反来对结果进行检查。

01 答案

$2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$ ，求反就是将这个数按位取反再加 1：

$$\begin{array}{r}
 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\
 + \hspace{10em} 1_2 \\
 \hline
 = \hspace{10em} 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 \\
 = \hspace{10em} -2_{10}
 \end{array}$$

另一方面，将

$$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$$

也按位取反再加 1：

⊖ 此公式从右往左第 3 项为 1×2^2 ，原书中为 1×2^1 ，原书有误。——译者注

$$\begin{array}{r}
 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\
 + \hspace{10em} \\
 \hline
 = \hspace{1em} 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 \\
 = \hspace{1em} 2_{10}
 \end{array}$$

77

1

第二种方法用于将一个用 n 位表示的二进制数转化成一个用多于 n 位表示的数。例如，在取数、存数、分支、加、小于则置位等指令中的立即数字段包含一个二进制补码表示的 16 位数，表示从 $-32\ 768_{10}$ (-2^{15}) 到 $32\ 767_{10}$ ($2^{15}-1$)。为了将这个立即数字段加到一个 32 位的寄存器，计算机必须将这个 16 位的数转换成数值上相等的 32 位的数。这种方法就是将原有的 16 位数简单复制到 32 位新数的低 16 位，其最高有效位（符号位）则以复制的方式填满新数的高 16 位。这种方法通常叫作符号扩展（sign extension）。

① 例题·符号扩展的方法

将 2_{10} 和 -2_{10} 从 16 位二进制数转换为 32 位二进制数。

答案

2_{10} 的 16 位二进制表示形式是

$$0000\ 0000\ 0000\ 0010_2 = 2_{10}$$

将这个数转化成 32 位数的方法是：将最高有效位（0）复制 16 次放到 32 位字的左半部。右半部的 16 位保持原 16 位的值：

$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 = 2_{10}$$

使用前面介绍的方法对 2 的 16 位二进制数求反。于是，

0000 0000 0000 0010₂

变成

$$\begin{array}{r}
 1111111111111101_2 \\
 + 1_2 \\
 \hline
 = 1111111111111110_2
 \end{array}$$

将该求反结果转换为 32 位数的方法就是将符号位复制 16 次放到 32 位字的左半部：

$$1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1111 \ 1110_2 = -2_{10}$$

这种方法之所以正确，是因为二进制补码表示的正数实际上在左侧有无限多个0，而负数在左侧有无限多个1。只是为了适应硬件的宽度，数的前导位被隐藏了，符号扩展只是简单地恢复了其中一部分。

78

小结

本节的主要内容是如何在给定的计算机字长中表示正整数和负整数。虽然各种表示方法都有各自的优缺点，但从 1965 年以来大多数计算机都采用了二进制补码方法。

01 精解 因为带符号十进制数没有长度的限制，所以常用“-”来表示负数。而在给定二进制或十六进制（见图 2-4）字长的情况下，可以将符号编码到位串中，因此通常不使用“+”和“-”来表示二进制或十六进制数。

01 小测验

下面这个 64 位二进制补码数对应的十进制数是多少？

- 1) -4_{10}
- 2) -8_{10}
- 3) -16_{10}
- 4) $18\ 446\ 744\ 073\ 709\ 551\ 609_{10}$

01 精解 二进制补码的得名来自下述规则：一个 n 位的数与它的相反数做无符号加法，结果是 $2n$ ，因此， x 的相反数 $-x$ 的二进制补码表示是 $2n - x$ 。

除了“二进制补码”和“符号和幅值”这两种表示法以外，第三种可选的表示法是所谓的“反码”(one's complement)。在反码中，一个数的相反数就是将这个数的每一位按位取反，0 变成 1，1 变成 0，这也是这种表示法名字的由来。在反码中 x 的相反数是 $2^n - x - 1$ 。与符号和幅值表示法相比，反码在某些方面是一个更好的解决方案，因此一些早期用于科学计算的计算机采用这种表示法。与补码相比，反码除了有 2 个零以外，其余都是相似的。其中正 0 是 $00\cdots00_2$ ，负 0 是 $11\cdots11_2$ 。绝对值最大的负数（即最小的负数）是 $10\cdots00_2$ ，它表示 $-2\ 147\ 483\ 647_{10}$ ，所以正数和负数的个数是平衡的。当采用反码时，加法器需要一个额外的步骤减去一个数来修正结果。因此，现在的计算机中补码方法占据了统治地位。

第 3 章将介绍一种浮点数的表示法。其中，最小的负数用 $00\cdots000_2$ 表示，最大的正数用 $11\cdots11_2$ 表示，0 一般用 $10\cdots00_2$ 表示。因为它通过将数加一个偏移使其具有非负的表示形式，所以称为偏移表示法 (biased notation)。

- ② 反码：使用 $10\cdots000_2$ 表示最小负数， $01\cdots11_2$ 表示最大正数，正数和负数的数量相同，但保留两个零，一个正零 ($00\cdots00_2$)，一个负零 ($11\cdots11_2$)。这种方法也用来表示按位求反，即 0 置为 1，1 置为 0。
- ③ 偏移表示法：最小的负数用 $00\cdots000_2$ 表示，最大的正数用 $11\cdots11_2$ 表示，0 一般用 $10\cdots00_2$ 表示，即通过将数加一个偏移使其具有非负的表示形式。

79

2.5 计算机中指令的表示

人操作计算机的方式与计算机看到指令的方式是不同的，现在我们就可以来解释其差别了。

指令在计算机内部是以若干或高或低的电信号的序列表示的，并且形式上和数的表示相同。实际上，指令的各部分都可看成一个独立的数，将这些数拼接在一起就形成了指令。

因为几乎所有的指令中都要用到寄存器，所以必须有一套规定，以将寄存器名字映射成数字。在 MIPS 汇编语言中，寄存器 $\$s0 \sim \$s7$ 映射到寄存器 $16 \sim 23$ ，同时，寄存器 $\$t0 \sim \$t7$ 映射到寄存器 $8 \sim 15$ 。因此， $\$s0$ 表示寄存器 16， $\$s1$ 表示寄存器 17， $\$s2$ 表示寄存器 18…… $\$t0$ 表示寄存器 8， $\$t1$ 表示寄存器 9，依次类推。在下面几节中，我们将介绍 32 个寄存器中其余寄存器的映射。

01 例题·将一条 MIPS 汇编语言指令翻译成一条机器指令

下面以 MIPS 汇编语言为例。对于符号表示为

`add $t0,$s1,$s2`

的 MIPS 指令，首先给出其十进制数表示形式，接着给出其二进制数表示形式。

01 答案

其十进制表示为

0	17	18	8	0	32
---	----	----	---	---	----

机器指令分为若干字段 (field)。本例中第一个字段和最后一个字段 (0 和 32) 组合起来告诉 MIPS 计算机该指令要完成加法运算。第二个字段表示加法的第一个源操作数寄存器号 ($17 = \$s1$)，第三个字段表示加法的另一个源操作数寄存器号 ($18 = \$s2$)。第四个字段表示存放运算结果的目的寄存器号 ($8 = \$t0$)。第五个字段在这条指令中没有用到，故置为 0。这样，这条指令将寄存器 $\$s1$ 和寄存器 $\$s2$ 内容相加，并将和放在寄存器 $\$t0$ 中。

这条指令也可以表示成二进制的形式：

000000	10001	10010	01000	00000	100000
6位	5位	5位	5位	5位	6位

80

□

指令的布局形式叫作指令格式 (instruction format)。从位的数目可以看出，MIPS 指令占 32 位，与数据字的位数相等。为遵循简单源于规整的原则，所有 MIPS 指令都是 32 位长。

为了将它与汇编语言区分开来，把指令的数字形式称为机器语言 (machine language)，这样的指令序列叫作机器码 (machine code)。

- ② 指令格式：二进制数字段组成的指令表示形式。
- ② 机器语言：在计算机系统中用于交流的二进制表示形式。

为避免读写冗长乏味的二进制字串，可采用比二进制基数更大，但又易转化为二进制的表示形式来表示。由于几乎所有计算机的数据大小都是 4 的整数倍，因此十六进制 (hexadecimal) 表示形式变得很流行。16 是 2 的 4 次幂，因此可以很简单地通过将每 4 位二进制数替换为 1 位十六进制数来完成二进制到十六进制的转换，反之亦然。图 2-4 给出了十六进制和二进制之间的转化表。

- ② 十六进制：基数为 16 的数。

十六进制	二进制	十六进制	二进制	十六进制	二进制	十六进制	二进制
0_{16}	0000_2	4_{16}	0100_2	8_{16}	1000_2	c_{16}	1100_2
1_{16}	0001_2	5_{16}	0101_2	9_{16}	1001_2	d_{16}	1101_2
2_{16}	0010_2	6_{16}	0110_2	a_{16}	1010_2	e_{16}	1110_2
3_{16}	0011_2	7_{16}	0111_2	b_{16}	1011_2	f_{16}	1111_2

图 2-4 十六进制和二进制转换表。可以简单地把 1 位十六进制数替换为相应的 4 位二进制数，反之亦然。如果二进制数的位数不是 4 的整数倍，转化要从右往左进行

为了避免处理不同进制数时产生混淆，此处约定十进制数加下标 10，二进制数加下标 2，十六进制数加下标 16。（如果没有下标，那么默认为十进制。）顺便说明，C 和 Java 中用符号 $0xnnnn$ 来表示十六进制数。

01 例题·二进制和十六进制间的转换

将下面的十六进制数转化成二进制数，二进制数转化成十六进制数：

$eca8\ 6420_{16}$

0001 0011 0101 0111 1001 1011 1101 1111₂

01 答案

按图 2-4 所示十六进制 - 二进制转换表查表得：



□

MIPS 字段

为了使讨论变得简单，给 MIPS 字段命名如下：

op	rs	rt	rd	shamt	funct
6位	5位	5位	5位	5位	6位

MIPS 指令中各字段名称及含义如下：

- op：指令的基本操作，通常称为操作码（opcode）。
- rs：第一个源操作数寄存器。
- rt：第二个源操作数寄存器。
- rd：用于存放操作结果的目的寄存器。
- shamt：位移量。（在 2.6 节中介绍移位指令和该术语，在此之前，指令都不使用这个字段，故此字段的内容为 0。）
- funct：功能。一般称为功能码（function code），用于指明 op 字段中操作的特定变式。

操作码：指令中用来表示操作和格式的字段。

当某条指令需要比上述字段更长的字段时，问题就会发生。例如，取字指令必须指定两个寄存器和一个常数。在上述格式中，如果地址使用其中的一个 5 位字段，那么取字指令的常数就被限制在 2^5 （即 32）之内。这个常数通常用来从数组或数据结构中选择元素，所以它常常比 32 大得多。5 位字段因太小而用处不大。

因此，既希望所有指令长度相同，又希望具有统一的指令格式，两者之间产生了冲突。这就引出了最后一条硬件设计原则。

设计原则 3：优秀的设计需要适宜的折中方案。

MIPS 设计者选择这样一种折中方案：保持所有的指令长度相同，但不同类型的指令采用不同的指令格式。例如，上述格式称为 R 型（用于寄存器）。另一种指令格式称为 I 型（用于立即数），立即数和数据传送指令用的就是这种格式。I 型的字段如下所示：

op	rs	rt	constant or address
6位	5位	5位	16位

16 位的地址字段意味着取字指令可以取相对于基址寄存器地址偏移 $\pm 2^{15}$ 或者 32 768 个字节 ($\pm 2^{13}$ 或者 8192 个字) 范围内的任意数据字。类似地，加立即数指令中常数也被限制不超过 $\pm 2^{15}$ 。可以看到在这种格式下，很难设置 32 个以上的寄存器，因为 rs 和 rt 字段都必须增加额外的位，这样就导致 32 位字长的指令很难满足要求。

我们来分析一下 2.3.1 节例子中的取字指令：

```
lw $t0,32($s3) # Temporary reg $t0 gets A[8]
```

这里，19（寄存器 \$s3）存放于 rs 字段，8（寄存器 \$t0）存放于 rt 字段，32 存放于 address 字段。注意，对于这条指令 rt 字段的意思已经改变：在一条取字指令中，rt 字段用于指明接收取数结果的目的寄存器。

虽然多种指令格式使硬件变得复杂，但是保持指令格式的类似性仍可降低复杂度。例如，R 型和 I 型格式的前 3 个字段长度相等，并且名称也一样；I 型格式的第四个字段和 R 型后 3 个字段长度之和相等。

也许你会想到，指令格式可以由第一个字段的值来区分：每种格式在第一个字段（op）占有不同的值区间，以便让计算机硬件知道指令后半部分是三字段（R 型）还是一字段（I 型）。图 2-5 给出了到目前为止已使用过的 MIPS 指令的每个字段的值。

指令	格式	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{10}	n. a.
sub (subtract)	R	0	reg	reg	reg	0	34_{10}	n. a.
add immediate	I	8_{10}	reg	reg	n. a.	n. a.	n. a.	常数
lw (load word)	I	35_{10}	reg	reg	n. a.	n. a.	n. a.	address
sw (store word)	I	43_{10}	reg	reg	n. a.	n. a.	n. a.	address

图 2-5 MIPS 指令编码。在上表中，“reg”代表寄存器的标号（从 0 ~ 31），“address”表示 16 位地址，“n. a.”（not applicable）表示这个字段在该指令格式中不出现。注意，add 和 sub 指令具有相同的 op 字段值，硬件根据 funct 字段的值来决定所进行的操作：add (32) 或 subtract (34)

83

01 例题·将 MIPS 汇编语言翻译成机器语言

现在可以给出一个例子来描述从程序员所编程序到机器执行指令的整个转换过程。如果数组 A 的基址存放在 \$t1 中，h 存放在 \$s2 中，下面的 C 赋值语句：

```
A[300] = h + A[300];
```

被编译成如下汇编语言：

```
lw $t0,1200($t1) # Temporary reg $t0 gets A[300]
add $t0,$s2,$t0    # Temporary reg $t0 gets h + A[300]
sw $t0,1200($t1) # Stores h + A[300] back into A[300]
```

这三条 MIPS 指令的机器语言代码是什么？

01 答案

为方便起见，先使用十进制数表示机器语言指令。从图 2-5 中可以确定这三条机器语言指令：

op	rs	rt	rd	address/shamt	funct
35	9	8		1200	
0	18	8	8	0	32
43	9	8		1200	

lw 指令的第一个字段（op）值为 35（见图 2-5）。在第二个字段（rs）中指定基址寄存器 9（\$t1），在第三个字段（rt）中指定目的寄存器 8（\$t0）。在最后一个字段 address 中存放用于指定 A [300] 的偏移量 ($1200 = 300 \times 4$)。

下一条 add 指令由第一个字段（op）值 0 和最后一个字段（funct）值 32 共同确定。第二、三、四字段中的三个寄存器（18、8 和 8）分别对应 \$s2、\$t0 和 \$t0。

`sw` 指令由第一个字段的 43 识别。这条指令的其他部分和 `lw` 指令完全一样。

与上述十进制形式对应的二进制机器指令如下所示（十进制数 1 200 用二进制表示为 0000 0100 1011 0000₂）：

100011	01001	01000	0000 0100 1011 0000		
000000	10010	01000	01000	00000	100000
101011	01001	01000	0000 0100 1011 0000		

注意，第一条指令和最后一条指令的二进制表示非常相似，唯一不同的是从左边数第 3 位。

84

01 硬件/软件接口 定长指令的需求与设置尽可能多的寄存器的需求矛盾。寄存器数量的任何增长都需要在指令格式中的各个寄存器字段至少增加 1 位。综合考虑这些限制和越小越快的设计原则，当今的大多数指令系统中有 16 个或 32 个通用寄存器。

图 2-6 归纳了本节讲述的 MIPS 机器语言。正如将在第 4 章中讲述的那样，相关指令在二进制表示上的相似性可简化硬件设计。这种相似性也是 MIPS 体系结构规整性的又一佐证。

MIPS 机器语言

名字	格式	举例						注释
add	R	0	18	19	17	0	32	add \$s1, \$s2, \$s3
sub	R	0	18	19	17	0	34	sub \$s1, \$s2, \$s3
addi	I	8	18	17	100			addi \$s1, \$s2, 100
lw	I	35	18	17	100			lw \$s1, 100(\$s2)
sw	I	43	18	17	100			sw \$s1, 100(\$s2)
字段宽度		6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令均为 32 位
R 型	R	op	rs	rt	rd	shamt	funct	算术指令格式
I 型	I	op	rs	rt	address			数据传送指令格式

图 2-6 2.5 节展示的 MIPS 体系结构。到目前为止所见到的 MIPS 指令都是 R 型和 I 型指令。所有指令的前 16 位都是相同的，都包含给出基本操作的 op 字段；给出第一源操作数的 rs 字段；给出第二源操作数的 rt 字段（取字指令除外，在取字指令中用于指定目的寄存器）。R 型指令将最后 16 位划分为 3 个字段：rd 字段指明目的寄存器；shamt 字段将在 2.6 节中介绍；funct 字段指明 R 型指令的特定辅助操作。I 型指令将最后 16 位合并为一个 address 字段

85

01 重点 当今计算机基于以下两个重要准则构建：

- 1) 指令用数的形式表示。
- 2) 和数据一样，程序存储在存储器中，并且可以读写。

这些原则引出存储程序（stored-program）的概念，这一发明释放了计算机的巨大潜力。图 2-7 显示了存储程序的强大功能。特别地，存储器可以存放编辑器程序的源代码、与之对应的编译后的机器码、编译后的程序需要使用的文本，甚至用于生成机器码的编译器。

指令表示成数的好处就是程序可以被当成二进制数的文件发行。商业上的意义就是计算机可以延用那些指令集兼容的现成软件。这种“二进制兼容”使得工业界围绕着几种指令集体系结构形成联盟。

86

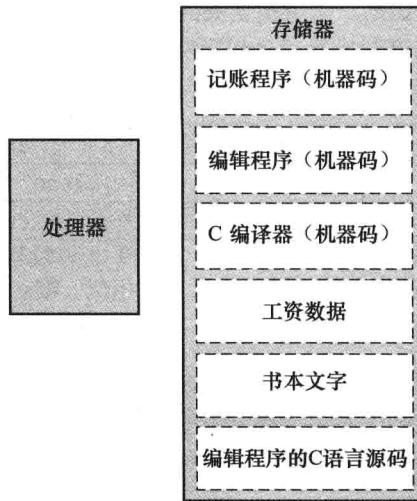


图 2-7 存储程序概念。各类存储程序允许将一台用于记账的计算机转眼间变成一台可以帮助作者写书的计算机。只要将程序和数据加载到存储器中并告诉计算机从给定的存储器地址开始执行程序即可。将指令和数据以相同的方式处理，极大地简化了计算机系统的存储器硬件和软件。尤其是用于数据的存储技术同样也适用于程序，如编译器，它能够将那些用易于人类使用的符号编写的代码翻译成机器能理解的代码。

01 小测验

下面的图表代表的是哪条 MIPS 指令？

op	rs	rt	rd	shamt	funct
0	8	9	10	0	34

1. sub \$t0, \$t1, \$t2
2. add \$t2, \$t0, \$t1
3. sub \$t2, \$t1, \$t0
4. sub \$t2, \$t0, \$t1

2.6 逻辑操作

“正相反，”叮当弟接着说，“如果那是真的，那它就可能是真的；如果那曾经是
真的，它就是真过；但是既然现在它不是真的，那么现在它就是假的。这就是逻辑。”

——Lewis Carroll, 《爱丽丝漫游仙境》, 1865

虽然早期的计算机仅对整字进行操作，但人们很快就发现，对字中由若干位组成的字段甚至对单个位进行操作是很有用的。例如，考查字里面每个由 8 位组成的字符（见 2.9 节）。于是，编程语言和指令集体系结构中增加了一些指令，用于简化对字中若干位进行打包或者拆包的操作。这些指令被称为逻辑操作。图 2-8 给出了 C、Java 和 MIPS 中的逻辑操作。

逻辑操作	C 操作符	Java 操作符	MIPS 指令
左移	<<	<<	sll
右移	>>	>>>	srl
按位与	&	&	and, andi
按位或			or, ori
按位取反	~	~	nor

图 2-8 C 和 Java 的逻辑操作符及相应的 MIPS 指令。MIPS 使用一个操作数为 0 的 NOR 指令实现取反操作

第一类逻辑操作称为移位 (shift)。它们将一个字里面的所有位都向左或向右移动，并在空出来的位上填充 0。例如，假设寄存器 \$s0 中的数据是：

0000 0000 0000 0000 0000 0000 1001₂ = 9₁₀

一条左移 4 位的指令执行后，得到的新值是：

0000 0000 0000 0000 0000 1001 0000₂ = 144₁₀

87

与左移相对应的是右移。左移和右移这两条指令在 MIPS 中的的确切名字是逻辑左移 (sll) 和逻辑右移 (srl)。下面的指令完成的就是上述操作，假设源操作数在 \$s0 中，结果存储到 \$t2 中：

sll \$t2,\$s0,4 # reg \$t2 = reg \$s0 << 4 bits

前面介绍 R 型指令格式时没有解释 shamt 字段，它在移位指令中被用于表示移位量 (shift amount)。因此，上述指令对应的机器语言是：

op	rs	rt	rd	shamt	funct
0	0	16	10	4	0

指令 sll 的编码在 op 字段和 funct 字段都为 0，rd 为 10（寄存器 \$t2），rt 为 16（寄存器 \$s0），shamt 为 4，rs 字段没有使用，被置为 0。

逻辑左移还有额外的好处，就是左移 i 位就相当于乘以 2^i ，这就像十进制数左移 i 位相当于乘以 10^i 。例如，上面的 sll 指令左移了 4 位，就相当于乘以 2^4 （即 16）。所以，原二进制数表示的值是 9，而 $9 \times 16 = 144$ ，恰好就是移位后的结果。

第二类有用的操作是按位与 (AND)。该操作仅当两个操作位均为 1 时结果才为 1。例如，如果寄存器 \$t2 的值为：

0000 0000 0000 0000 1101 1100 0000₂

寄存器 \$t1 的值为：

0000 0000 0000 0000 0011 1100 0000 0000₂

那么，在执行下面的 MIPS 指令后

and \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 & reg \$t2

\$t0 中的值将是：

0000 0000 0000 0000 1100 0000 0000₂

AND 提供了一种将源操作数中某些位置为 0 的能力，前提是另一个操作数中对应位为 0。后一个操作数传统上被称为掩码 (mask)，寓意其可“隐藏”某些位。

88

与 AND 对偶的操作是按位或 (OR)。该操作在两个操作位中任意一位为 1 时结果就为 1。为详细说明，仍假设 \$t1 和 \$t2 中的值都和上面的例子一样，那么下述 MIPS 指令

or \$t0,\$t1,\$t2 # reg \$t0 = reg \$t1 | reg \$t2

执行后 \$t0 的值是：

0000 0000 0000 0000 0011 1101 1100 0000₂

最后一类逻辑操作是按位取反 (NOT)。该操作仅有一个操作数，将 1 变成 0，0 变成 1。使用前面的符号，它可用来计算 \bar{x} 。为了保持三操作数的格式，MIPS 的设计者引入或非 NOR (NOT OR) 指令来取代 NOT。如果一个操作数是 0，那么对另一个操作数而言，结果就等价于 NOT: A NOR 0 = NOT (A OR 0) = NOT (A)。

如果寄存器 \$t1 中的值与上例保持不变，寄存器 \$t3 中的值是 0，那么下面 MIPS 指令

nor \$t0,\$t1,\$t3 # reg \$t0 = ~ (reg \$t1 | reg \$t3)

在寄存器 \$t0 中的执行结果是：

1111 1111 1111 1111 1100 0011 1111 1111₂

- ② 按位与：按位进行与操作，仅当两个操作位均为 1 时结果才为 1。
- ③ 按位或：按位进行或操作，当两个操作位中任意一位为 1 时结果就为 1。
- ④ 按位取反：按位进行非操作，仅有一个操作数，将 1 变成 0，0 变成 1。
- ⑤ 或非：按位先或后非操作，仅当两个操作位均为 0 时结果才为 1。

图 2-8 显示了 C 和 Java 的操作符与 MIPS 指令之间的关系。像在算术运算中一样，常数在 AND 和 OR 这些逻辑运算里也是很有用的，因此 MIPS 也提供了立即数与 (andi) 和立即数或 (ori) 指令。常数在 NOR 中出现得很少，因为 NOR 主要功能就是将单操作数按位取反，因此，MIPS 指令集体系结构没有设计支持 NOR 立即数的版本。

- 01 精解** MIPS 指令全集也包括异或 (XOR)，当两个操作数对应位不同时置 1，相同时置 0。C 语言允许在字内定义由若干位组成的一个或多个字段，并将其作为对象包装在一个字内，以适应如 I/O 设备等的外部接口需求。所有字段必须放在一个单字之中，并采用无符号整数。C 编译器使用 MIPS 的下列逻辑指令插入和提取字段：and、or、sll 以及 srl。
- 01 精解** 在与立即数进行逻辑与操作和逻辑或操作时，立即数的高 16 位补 0 后形成 32 位常数进行计算，而与立即数做加法运算时，将立即数进行符号扩展。

01 小测验

下面哪个操作可以将字中的一部分分离出来？

1. AND
2. 左移后再进行右移

89

2.7 决策指令

自动化计算机的实用性取决于重复使用给定指令序列的可能性，重复的次数取决于计算的结果……这一选择可以根据数的符号来决定（计算机认为 0 是正数）。因此，我们引入一条“指令”（条件转移“指令”），它根据给定数的符号从两条路径中选择正确的一条来执行。

——Burks、Goldstine、von Neumann, 1947

计算机与简单计算器的区别在于决策能力。根据输入数据和计算过程中产生的值，它可以执行不同的指令。程序语言通常使用 if 语句描述决策，有时也使用 go to 语句和标签。MIPS 汇编语言中有两条类似 if 和 go to 语句功能的指令。第一条是

beq register1, register2, L1

该指令表示：如果 register1 和 register2 中的数值相等，则转到标签为 L1 的语句执行。助记符 beq 代表如果相等则分支 (branch if equal)。

第二条指令是

bne register1, register2, L1

该指令表示：如果 register1 和 register2 中的数值不相等，则转到标签为 L1 的语句执行。助记符 bne 代表如果不相等则分支 (branch if not equal)。这两条指令传统上称为条件分支 (conditional branch) 指令。

- ② 条件分支：该指令先比较两个值，然后根据比较的结果决定是否从程序中的一个新地址开始执行指令序列。

01 例题·将 if-then-else 语句编译成条件分支指令

在下面这段代码中，`f`、`g`、`h`、`i`、`j` 都是变量，设该 5 个变量依次对应于从 `$s0` 到 `$s4` 的寄存器，求这条 C 语言 if 语句编译后形成的 MIPS 代码。

```
if (i == j) f = g + h; else f = g - h;
```

01 答案

图 2-9 是 MIPS 代码执行过程的流程图。第一个表达式比较 `i` 和 `j` 是否相等，需要一条 `beq` 指令。通常，通过测试分支的相反条件来跳过 if 语句后面的 `then` 部分，代码的效率会更高（标签 `Else` 将在后面定义）所以我们使用 `bne` 指令：

```
bne $s3,$s4,Else    # go to Else if i ≠ j
```

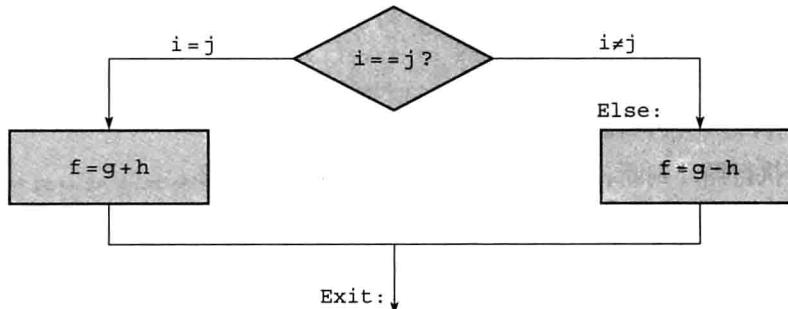


图 2-9 上述 if 语句的程序流程图。左边方框对应 if 语句的 then 部分，右边方框对应 if 语句的 else 部分

下一个赋值语句执行一个单操作，如果所有的操作数都分配给寄存器，那么它只是一条指令：

```
add $s0,$s1,$s2    # f = g + h (skipped if i ≠ j)
```

在 if 语句的结尾部分，需要引入另一种分支指令，通常叫作无条件分支指令（unconditional branch）。当遇到这种指令时，程序必须分支。为了区分条件分支和无条件分支，MIPS 将无条件分支指令命名为 `jump`，简写成 `j`（标签 `Exit` 将在后面定义）。

```
j Exit      # go to Exit
```

if 语句中 else 部分的赋值语句也可编译成一条指令。我们只需将标签 `Else` 加在这条指令前、标签 `Exit` 加在该条指令后面，表示 if-then-else 编译的代码结束：

```
Else:sub $s0,$s1,$s2    # f = g - h (skipped if i = j)
Exit:
```

注意，就像汇编器完成存数/取数指令的数据地址计算一样，它也完成分支指令的地址计算，这使得编译器和汇编语言程序员摆脱了乏味的地址计算任务（参见 2.12 节）。

01 硬件/软件接口 编译器经常创建一些在编程语言中没出现过的分支和标签。避免显式地编写这些标签和分支是使用高级编程语言的好处之一，也是其编码速度快的一个原因。

2.7.1 循环

无论是在二选一的 if 语句中，还是在迭代计算的循环语句中，决策都起着重要作用。但这两种情况下，关于决策的汇编语言指令是相同的。

01 例题·编译下面 C 语言 while 循环语句

下面是用 C 语言编写的传统循环程序：

```
while (save[i] == k)
    i += 1;
```

假设 i 和 k 存放在寄存器 $\$s3$ 和 $\$s5$ 中，数组 $save$ 的基址存放在寄存器 $\$s6$ 中。求这段 C 程序对应的 MIPS 汇编代码。

01 答案

第一步需要将 $save[i]$ 读入一个临时寄存器中。在读入之前，需要计算它的地址。在将 i 加到 $save$ 数组基址以形成访存地址前，由于系统按照字节寻址的缘故，先要将 i 乘以 4。幸运的是，我们可以使用逻辑左移指令实现这一乘法，因为左移 2 位等价于乘 4（见 2.6 节）。需要在该指令前增加一个标签 Loop，以便在循环末端能够跳回该指令。

```
Loop: sll $t1,$s3,2      # Temp reg $t1 = i * 4
```

为了得到 $save[i]$ 的地址，需要将 $\$t1$ 和 $\$s6$ 中 $save$ 的基址相加：

```
add $t1,$t1,$s6      # $t1 = address of save[i]
```

现在可用该地址将 $save[i]$ 读入一个临时寄存器中：

```
lw $t0,0($t1)        # Temp reg $t0 = save[i]
```

下一条指令执行循环判断，如果 $save[i] \neq k$ 则退出循环：

92 bne \$t0,\$s5, Exit # go to Exit if save[i] ≠ k

再下一条指令将 i 加 1：

```
addi $s3,$s3,1        # i = i + 1
```

在循环的末尾，程序跳转到循环的开始。随后增加了一个 Exit 标签，这样就完成了全部编译：

```
j      Loop          # go to Loop
Exit:
```

（见练习题中对该指令序列的优化。）

□

01 硬件/软件接口 以分支指令结束的这类指令序列对编译非常重要，因此它们有对应的专用术语：基本块。基本块（basic block）是没有分支（可能出现在末尾者除外）并且没有分支目标/分支标签（可能出现在开始者除外）的指令序列。编译最初阶段的任务之一就是将程序分解为若干基本块。

93 ② 基本块：没有分支（可能出现在末尾者除外）并且没有分支目标/分支标签（可能出现在开始者除外）的指令序列。

最常见的判断语句可能是相等或不等，但有时判断一个变量是否小于另一个变量也非常有用。例如，for 循环就需要判断索引变量是否小于 0。在 MIPS 汇编语言中提供了一条指令来实现这种比较，该指令在比较两个寄存器内容之后，若第一个寄存器小于第二个寄存器，则将第三个寄存器设置为 1，否则设置为 0。该指令称为小于则置位（set on less than），即 `slt`。例如，

```
slt      $t0, $s3, $s4    # $t0 = 1 if $s3 < $s4
```

表示当寄存器 $\$s3$ 的值小于寄存器 $\$s4$ 的值时，寄存器 $\$t0$ 被置为 1，否则寄存器 $\$t0$ 被置为 0。

在比较中经常使用常数操作数，所以有立即数版本的小于则置位指令。例如，为了测试寄存器 $\$s2$ 的值是否小于常数 10，可以使用如下指令：

```
slti     $t0,$s2,10      # $t0 = 1 if $s2 < 10
```

01 硬件/软件接口 MIPS 编译器使用 `slt`、`slti`、`beq`、`bne` 和固定值 0（总是可以通过读

取寄存器 \$zero 来获得) 来创建所有的比较条件：相等、不等、小于、小于或等于、大于、大于或等于。

遵循冯·诺伊曼关于“设备”简单性的原则，MIPS 体系结构没有提供“小于则分支”指令，因为这种指令过于复杂，它会延长时钟周期时间，或增加平均执行每条指令的周期数(CPI)。两条更快的指令更加有用。

01 硬件/软件接口 比较指令应该具有分清有符号数和无符号数的能力。有时候二进制数最高位为 1 的数代表一个负数，它当然应该小于所有最高有效位为 0 的正数。另一方面，如果是无符号数，最高有效位为 1 的数将大于所有最高有效位为 0 的数。(我们将很快看到最高有效位具有双重意义在减少数组边界检查开销中所带来的优点。)

MIPS 为这两种情况提供两个版本的小于则置位指令。slt (set on less than) 和 slti (set on less than immediate) 指令用于处理有符号整数，而 sltu (set on less than unsigned) 和 sltiu (set on less than immediate unsigned) 指令则用于处理无符号整数。

01 例题·有符号比较和无符号比较的对比

假设寄存器 \$s0 中的二进制数为

1111 1111 1111 1111 1111 1111 1111 1111₂

而寄存器 \$s1 中的二进制数为

0000 0000 0000 0000 0000 0000 0000 0001₂

在执行以下两条指令后，寄存器 \$t0 和 \$t1 中的值分别是多少？

```
slt      $t0, $s0, $s1 # signed comparison
sltu    $t1, $s0, $s1 # unsigned comparison
```

01 答案

如果是有符号数，那么寄存器 \$s0 中的值为 -1_{10} ，寄存器 \$s1 中的值为 1_{10} ；如果是无符号数，那么寄存器 \$s0 中的值为 $4\ 294\ 967\ 295_{10}$ ，寄存器 \$s1 中的值仍为 1_{10} 。因此，寄存器 \$t0 中的值为 1，因为 $-1_{10} < 1_{10}$ ；寄存器 \$t1 中的值为 0，因为 $4\ 294\ 967\ 295_{10} > 1_{10}$ 。 □

将有符号数作为无符号数来处理，是一种检验 $0 \leq x < y$ 的低开销方法，常用于检查数组的下标是否越界。问题的关键是负数在二进制补码表示法中看起来像是无符号表示法中一个很大的数，因为在无符号数中最高有效位是符号位，而有符号数中最高有效位是具有最大权重的位。所以使用无符号比较 $x < y$ ，在检查 x 是否小于 y 的同时，也检查了 x 是否为一个负数。

01 例题·边界检查的简便方法

利用这个方法可以降低检验下标是否越界的开销：如果 $\$s1 \geq \$t2$ 或者 $\$s1$ 是负数则跳转到 IndexOutOfBounds。

01 答案

检查代码仅使用一条 sltu 指令即可同时进行两种检查：

```
sltu $t0,$s1,$t2 # $t0=0 if $s1>=length or $s1<0
beq  $t0,$zero,IndexOutOfBounds #if bad, goto Error
```

□

2.7.2 case/switch 语句

大多数程序设计语言中都包括 case 或 switch 语句，使得程序员可以根据某个变量的值选择

不同分支之一。实现 switch 语句的最简单方法是借助一系列的条件判断，将 switch 语句转化为 if-then-else 语句嵌套。

有时候另一种更有效的方法是将多个指令序列分支的地址编码为一张表，即转移地址表 (jump address table) 或转移表 (jump table)，这样程序只需索引该表即可跳转到恰当的指令序列。转移地址表是一个由代码中标签所对应地址构成的数组。程序需要跳转的时候首先将转移地址表中适当的项加载到寄存器中，然后使用寄存器中的地址值进行跳转。为了支持这种情况，像 MIPS 这样的计算机提供了寄存器跳转指令 `jr` (jump register)，用来无条件地跳转到寄存器指定的地址。该指令将在下一节中介绍。

⑤ 转移地址表：又称作转移表 (jump table)，指包含不同指令序列地址的表。

01 硬件/软件接口 虽然在 C 或 Java 这样的编程语言中有许多决策和循环语句，但是在指令集这一层次实现其功能的基本语句是条件分支。

01 精解 如果你曾经听说过延迟转移（将在第 4 章中介绍），那么不必对此表示担心：MIPS 汇编器会使其对汇编语言程序员不可见。

01 小测验

I. C 语言中有很多决策和循环语句，但是在 MIPS 中却很少。下述各句子有没有阐明这种不均衡？为什么？

1. 更多的决策语句使得代码更容易被阅读和理解。
2. 更少的决策语句简化了负责执行的底层工作。
3. 更多的决策语句意味着更少的代码量，这节约了编程的时间。
4. 更多的决策语句意味着更少的代码量，这意味着执行更多的操作。

II. 为什么 C 语言提供了两种与操作 (& 和 &&) 和两种或操作 (| 和 ||)，而 MIPS 没有提供呢？

1. 逻辑操作 AND 和 OR 实现 & 和 |，而条件分支实现 && 和 ||。
2. 上面的描述说反了：&& 和 || 对应于逻辑操作，而 & 和 | 对应于条件分支。
3. 它们是冗余的，并且是一回事。&& 和 || 都是简单继承于 C 程序设计语言的前身：B 语言。

2.8 计算机硬件对过程的支持

过程 (procedure) 或函数是程序员进行结构化编程的工具，两者均有助于提高程序的可理解性和代码的可重用性。过程允许程序员每次只需将精力集中在任务的一部分，由于参数能传递数值并返回结果，因此参数承担过程与其他程序、数据之间接口的角色。2.15 节描述了 Java 语言中过程的等价表示方法，但 Java 与 C 语言对计算机的要求完全相同。过程是软件中实现抽象的一种方法。

⑥ 过程：根据提供的参数执行一定任务的存储的子程序。

你可以将过程想象成一个侦探，他离开时带着一项神秘的计划，为了完成该计划，需要获得资源、执行任务并隐匿行踪，最后带着预期的结果返回起点。一旦任务完成将不再对系统产生任何其他干扰。更重要的是，侦探是在“需要知道”的基础上工作的，所以侦探不需对雇主做任何假定。

同样，在过程运行中，程序必须遵循以下 6 个步骤：

- 1) 将参数放在过程可以访问的位置。
- 2) 将控制转交给过程。
- 3) 获得过程所需的存储资源。
- 4) 执行需要的任务。
- 5) 将结果的值放在调用程序可以访问的位置。
- 6) 将控制返回初始点，因为一个过程可能由一个程序中的多个点调用。

如上所述，寄存器是计算机中保存数据最快的位置，所以我们希望尽可能多地使用寄存器。MIPS 软件在为过程调用分配 32 个寄存器时遵循以下约定：

- \$a0 ~ \$a3：用于传递参数的 4 个参数寄存器。
- \$v0 ~ \$v1：用于返回值的两个值寄存器。
- \$ra：用于返回起始点的返回地址寄存器。

除了分配这些寄存器之外，MIPS 汇编语言还包括一条过程调用指令：跳转到某个地址的同时将下一条指令的地址保存在寄存器 \$ra 中。这条跳转和链接指令（jump-and-link instruction）的格式为：

```
jal ProcedureAddress
```

指令中的链接部分表示指向调用点的地址或链接，以允许过程返回到合适的地址。存储在寄存器 \$ra (31 号寄存器) 中的链接部分称为返回地址 (return address)。返回地址是必需的，因为同一过程可能在程序的不同部分调用。

- 跳转和链接指令：跳转到某个地址的同时将下一条指令的地址保存到寄存器 \$ra 中的指令。
- 返回地址：指向调用点的链接，使过程可以返回到合适的地址，在 MIPS 中它存储在寄存器 \$ra 中。

为了支持这种情况，类似 MIPS 的计算机使用了寄存器跳转 (jump register) 指令 jr，用于 case 语句，表示无条件跳转到寄存器所指定的地址：

```
jr $ra
```

寄存器跳转指令跳转到存储在 \$ra 寄存器中的地址——这正是我们所希望的。因此，调用程序或称为调用者 (caller)，将参数值放在 \$a0 ~ \$a3，然后使用 jal x 跳转到过程 x (有时称为被调用者 (callee))。被调用者执行运算，将结果放在 \$v0 和 \$v1，然后使用 jr \$ra 指令将控制返回给调用者。

在存储程序概念中，使用一个寄存器来保存当前运行的指令地址是绝对必要的。尽管这个寄存器更为合理的名字可能应该是指令地址寄存器 (instruction address register)，但是出于历史原因，这个寄存器通常称为程序计数器 (program counter)，在 MIPS 体系结构中缩写为 PC。jal 指令实际上将 PC + 4 保存在寄存器 \$ra 中，从而将链接指向下一条指令，为过程返回做好准备。

- 调用者：调用一个过程并为过程提供必要参数值的程序。
- 被调用者：根据调用者提供的参数执行一系列存储的指令，然后将控制权返回调用者的程序。
- 程序计数器 (PC)：包含在程序中正在被执行指令地址的寄存器。

2.8.1 使用更多的寄存器

假设对于一个过程，编译器需要使用多于 4 个参数寄存器和两个返回值寄存器。由于在任务完成后必须消除踪迹，因此调用者使用的任何寄存器都必须恢复到过程调用前所存储的值。这种情况可以看成是需要将寄存器换出到存储器的一个例子，如“硬件/软件接口”部分所提到的那样。

换出寄存器的最理想的数据结构是 **栈** (stack) ——一种后进先出的队列。栈需要一个指针指向栈中最新分配的地址，以指示下一个过程放置换出寄存器的位置，或是寄存器旧值的存放位置。在每次寄存器进行保存或恢复时，**栈指针** (stack pointer) 以字为单位进行调整。MIPS 软件为栈指针准备了第 29 号寄存器，并将其命名为 \$sp。由于栈的应用十分广泛，因此向栈传递数据或从栈中取数都有专用术语：将数据放入栈中称为 **压栈** (push)，从栈中移除数据称为 **出栈** (pop)。

- ② 栈：被组织成后进先出队列形式并用于寄存器换出的数据结构。
- ② 栈指针：指示栈中最近分配的地址的值，它指示寄存器被换出的位置，或寄存器旧值的存放位置。在 MIPS 中，栈指针是寄存器 \$sp。
- ② 压栈：向栈中增加元素。
- ② 出栈：从栈中移除元素。

按照历史惯例，栈“增长”是按照地址从高到低的顺序进行的。这意味着将数据压栈时，栈指针值减小；而数据出栈时，栈长度缩短，栈指针增大。

01 例题·编译一个不调用其他过程的 C 过程

将 2.2 节的例子转化为一个 C 过程：

```
int leaf_example (int g, int h, int i, int j)
{
    int f;

    f = (g + h) - (i + j);
    return f;
}
```

98

编译后的 MIPS 汇编代码是什么呢？

01 答案

参数变量 g、h、i 和 j 对应参数寄存器 \$a0、\$a1、\$a2 和 \$a3，f 对应 \$s0。编译后的程序是以如下标号开始的过程：

`leaf_example:`

下一步是保存过程中使用的寄存器。过程实体中的 C 赋值语句与 2.2 节的例子相同，使用了两个临时寄存器。因此，需要保存三个寄存器：\$s0、\$t0 和 \$t1。我们将旧值“压栈”，也就是在栈中建立三个字的空间，并将数据存入：

```
addi $sp, $sp, -12    # adjust stack to make room for 3 items
sw    $t1, 8($sp)     # save register $t1 for use afterwards
sw    $t0, 4($sp)     # save register $t0 for use afterwards
sw    $s0, 0($sp)     # save register $s0 for use afterwards
```

图 2-10 给出了在过程调用之前、之中和之后的栈。

接着的三条语句对应过程实体，与 2.2 节的例子相同：

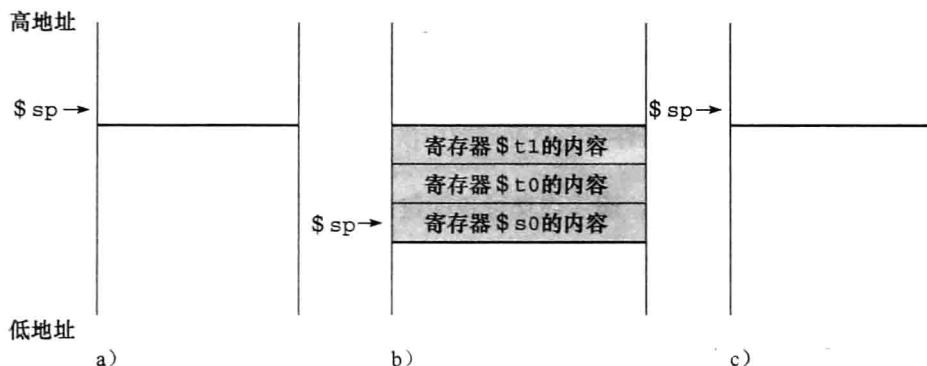


图 2-10 在过程调用之前 (a)、之中 (b) 和之后 (c)，栈指针以及栈的值。栈指针总是指向栈顶，或者说是图中栈的最后一个字

```
add $t0,$a0,$a1 # register $t0 contains g + h
add $t1,$a2,$a3 # register $t1 contains i + j
sub $s0,$t0,$t1 # f = $t0 - $t1, which is (g + h)-(i + j)
```

为了返回 f 的值，我们将它复制到一个返回值寄存器中：

```
add $v0,$s0,$zero # returns f ($v0 = $s0 + 0)
```

在返回前，我们通过从栈中“弹出”数据的方式恢复寄存器的三个旧值：

```
lw $s0, 0($sp) # restore register $s0 for caller
lw $t0, 4($sp) # restore register $t0 for caller
lw $t1, 8($sp) # restore register $t1 for caller
addi $sp,$sp,12 # adjust stack to delete 3 items
```

过程末尾处根据跳转寄存器中的返回地址跳转：

```
jr $ra # jump back to calling routine
```

□

前面的例子曾经使用了临时寄存器，并假设它们的旧值必须保存和恢复。为了避免保存和恢复一个其值未被使用过的寄存器（通常是临时寄存器），MIPS 软件将 18 个寄存器分为两组：

- \$t0 ~ \$t9：10 个临时寄存器，在过程调用中不必被调用者（被调用的过程）保存。
- \$s0 ~ \$s7：8 个保留寄存器，在过程调用中必须被保存（一旦被使用，由被调用者保存和恢复）。

99

这一简单约定减少了寄存器换出。在上面的例子中，因为调用者不希望在过程调用时保留寄存器 \$t0 和 \$t1，所以我们可以去掉有关两次保存和两次载入的代码。我们始终需要保存和恢复 \$s0，因为被调用者必须假设调用者需要该值。

2.8.2 嵌套过程

不调用其他过程的过程称为叶过程 (leaf procedure)。如果所有过程都是叶过程，那么情况就很简单，但实际并非如此。就像一个侦探，其任务的一部分是雇用其他侦探，被雇用的侦探进而雇用更多的侦探，某个过程调用其他过程也是这样。更进一步的是，递归过程甚至调用的是自身的“克隆”。就像在过程中使用寄存器需要十分小心一样，在调用非叶过程时需要更加小心。

例如，假设主程序将参数 3 存入寄存器 \$a0，然后使用 jal A 调用过程 A。再假设过程 A 通过 jal B 调用过程 B，参数为 7，同样存入 \$a0。由于 A 尚未结束任务，所以在寄存器 \$a0 的使用上存在冲突。同样，在寄存器 \$ra 保存的返回地址上也存在冲突，因为它现在保存着 B 的返回地址。除非我们采取措施阻止这类问题发生，否则这个冲突将导致过程 A 无法返回其调用者。

一个解决方法是将其他所有必须保留的寄存器压栈，就像将保存寄存器压栈一样。调用者将所有调用后还需要的参数寄存器（\$a0 ~ \$a3）或临时寄存器（\$t0 ~ \$t9）压栈。被调用者将返回地址寄存器 \$ra 和被调用者使用的保存寄存器（\$s0 ~ \$s7）都压栈。栈指针 \$sp 随着栈中寄存器个数调整。到返回时，寄存器会从存储器中恢复，栈指针也随着重新调整。

100

01 例题·编译一个递归 C 过程，演示嵌套过程的链接

下面是一个计算阶乘的递归过程：

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

该过程的 MIPS 汇编代码是怎样的呢？

01 答案

参变量 n 对应参数寄存器 \$a0。编译后的程序以过程标签开始，然后在栈中保存两个寄存器，一个是返回地址，另一个是 \$a0：

```
fact:
    addi $sp, $sp, -8 # adjust stack for 2 items
    sw $ra, 4($sp) # save the return address
    sw $a0, 0($sp) # save the argument n
```

第一次调用 fact 时，sw 保存程序中调用 fact 的地址。紧接着的两条指令测试 n 是否小于 1，如果 $n \geq 1$ 则跳转到 L1。

```
slti $t0,$a0,1 # test for n < 1
beq $t0,$zero,L1 # if n >= 1, go to L1
```

如果 n 小于 1，fact 将 1 置入一个值寄存器并返回。具体做法是在 0 上加 1 再将和存入 \$v0。然后从栈中弹出两个已保存的值并跳转到返回地址：

```
addi $v0,$zero,1 # return 1
addi $sp,$sp,8 # pop 2 items off stack
jr $ra # return to caller
```

在从栈中退出两项之前，本应该加载 \$a0 和 \$ra。但由于 n 小于 1 时，\$a0 和 \$ra 没有变化，所以就跳过了这些指令。

如果 n 不小于 1，参数 n 减 1 后，使用减 1 后的值再次调用 fact：

```
L1: addi $a0,$a0,-1 # n >= 1: argument gets (n - 1)
    jal fact # call fact with (n - 1)
```

下一条指令是 fact 的返回位置。现在旧的返回地址和旧的参数以及栈指针都需要恢复：

```
lw $a0, 0($sp) # return from jal: restore argument n
lw $ra, 4($sp) # restore the return address
addi $sp, $sp, 8 # adjust stack pointer to pop 2 items
```

接下来，值寄存器 \$v0 得到旧参数 \$a0 和当前值寄存器的乘积。这里假设乘法指令是可用的，尽管直到第 3 章才涉及乘法指令。

```
mul $v0,$a0,$v0 # return n * fact (n - 1)
```

最后，fact 再次跳转到返回地址：

```
jr $ra # return to the caller
```

□

01 硬件/软件接口 C 语言中的一个变量通常对应存储中的一个位置，其解释取决于其类型（type）和存储方式（storage class）。这方面的例子包括整型和字符型（见 2.9 节）。C 语言包括两种存储方式：动态的（automatic）和静态的（static）。动态变量位于过程中，当

过程退出时失效。静态变量在进入和退出过程时始终存在。在所有过程之外声明的 C 变量，以及声明时使用关键字 static 的变量都被视作静态的，其余的变量都被视作动态的。为了简化静态数据的访问，MIPS 软件保留了另一个寄存器，称为全局指针（global pointer），即 \$gp。

● 全局指针：指向静态数据区的保留寄存器。

图 2-11 总结了过程调用时所需保存的内容。需要注意的是，一些方案保存了栈，以确保调用者出栈时得到与压栈时相同的数据。只需保证被调用者不在 \$sp 以上进行写操作，\$sp 以上的栈就可以得到保存；而 \$sp 本身的保存是通过按被调用者将减去值的相同数量重新加上来实现的，其他寄存器则通过将它们保存到栈（如果它们被使用到的话）再从栈中恢复它们来进行保存。

保留	不保留
保存寄存器：\$s0 ~ \$s7	临时寄存器：\$t0 ~ \$t9
栈指针寄存器：\$sp	参数寄存器：\$a0 ~ \$a3
返回地址寄存器：\$ra	返回值寄存器：\$v0 ~ \$v1
栈指针以上的栈	栈指针以下的栈

图 2-11 过程调用时，保留和不保留的内容。如果软件依赖于下面将讨论的帧指针寄存器或者全局指针寄存器，那么它们也需要保留

102

2.8.3 在栈中为新数据分配空间

栈的最后一点复杂性是栈还需要存储过程的局部变量，但这些变量不适用于寄存器，例如局部的数组或结构体。栈中包含过程所保存的寄存器和局部变量的片段称为过程帧（procedure frame）或活动记录（activation record）。图 2-12 显示了过程调用之前、之中和之后栈的状态。

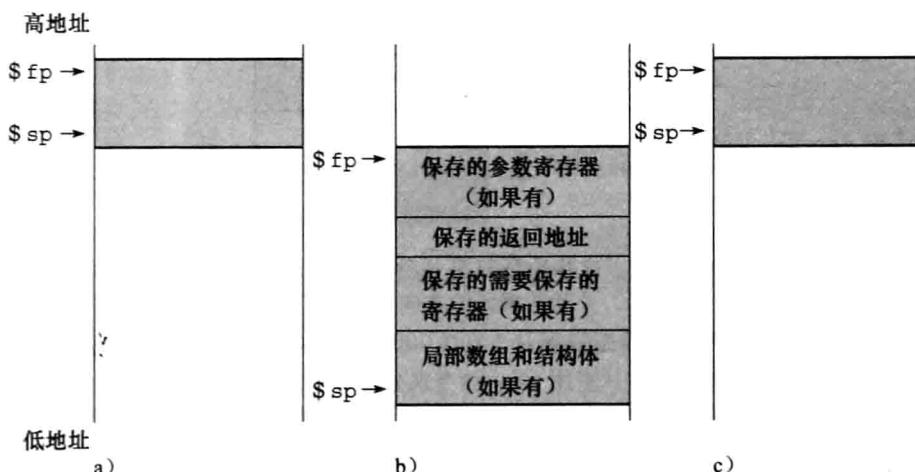


图 2-12 过程调用之前 (a)、之中 (b)、之后 (c) 栈的分配情况。帧指针 (\$fp) 指向该帧的第一个字（一般是保存的参数寄存器），而栈指针 (\$sp) 指向栈顶。栈可调整为有足够的空间来容纳所有的保存寄存器和驻留内存的局部变量。因为在程序运行期栈指针可能会改变，所以对于程序员而言，虽然使用栈指针和少量的地址运算就可能完成对变量的引用，但使用固定的帧指针引用变量会更为简单。如果在一个过程中栈内没有局部变量，编译器将可以不设置和不恢复帧指针以节省时间。当使用帧指针时，在调用中使用 \$sp 的地址进行初始化，而 \$sp 可以使用 \$fp 来恢复。相关内容可以在 MIPS 参考数据卡的第 4 列找到

103

某些 MIPS 软件使用帧指针 (frame pointer, \$fp) 指向过程帧的第一个字。在过程中栈指针可能会发生改变，因此存储器中对局部变量的引用在过程中的不同位置可能具有不同的偏移量，这使得过程更加难以理解。另一种方案，帧指针在一个过程中为局部存储器引用提供一个固定的基址寄存器。注意，无论是否使用显式的帧指针，活动记录都出现在栈中。我们通过避免在过程中修改 \$sp 来避免使用 \$fp，在我们的例子中，栈只在过程的入口和出口需要调整。

- ② 过程帧：也称作活动记录，栈中包含过程所保存的寄存器以及局部变量的片段。
- ③ 帧指针：指向给定过程中保存的寄存器和局部变量的值。

2.8.4 在堆中为新数据分配空间

除了动态变量对过程是局部有效之外，C 程序员还需要在内存中为静态变量和动态数据结构提供空间。图 2-13 给出了 MIPS 分配内存的约定。栈由内存高端开始并向向下增长。内存低端的第一部分是保留的，之后是 MIPS 机器代码的第一部分，通常称为代码段 (text segment)。代码段之上的代码为静态数据段 (static data segment)，是存储常量和其他静态变量的空间。尽管数组通常具有固定长度因而能与静态数据段很好地匹配，但类似链表这样的数据结构通常会在生命期内增长或缩短。这类数据结构对应的段习惯上称为堆 (heap)，一般在存储器中放在静态数据段之后。注意，这种分配允许栈和堆相互生长，从而在两个段此消彼长的过程中达到内存的高效使用。

- ② 代码段：UNIX 目标文件中的段，包含源文件中例程对应的机器语言代码。

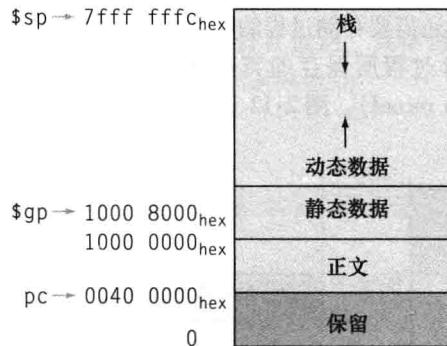


图 2-13 程序和数据的 MIPS 内存分配。这些地址只是一种软件规定，并非 MIPS 体系结构的一部分。栈指针初始化为 7fff ffff₁₆，并朝数据段的方向向下增长。在另一端，程序代码（代码段）从地址 0040 0000₁₆ 开始。静态数据从 1000 0000₁₆ 开始。然后是动态数据，在 C 中使用 malloc 命令分配，在 Java 中使用 new 命令来分配。动态数据在某一区域中朝着栈的方向向上生长，该区域称为堆。全局指针 \$gp 应设置为适当地址以便于访问数据。它初始化为 1000 8000₁₆，这样通过相对 \$gp 的正负 16 位的偏移量就可以访问从 1000 0000₁₆ 到 1000 ffff₁₆ 之间的内存空间。关于这点可参见 MIPS 参考数据卡的第 4 列。

C 语言通过显式的函数调用在堆上分配和释放空间。malloc() 在堆上分配空间并返回指向它的指针，free() 释放指针指向的堆空间。内存分配由 C 程序控制，这是很多错误产生的根源。忘记释放空间会导致“内存泄漏”，它会逐渐耗尽大量内存以至于操作系统可能崩溃。过早释放空间会导致“悬摆指针”(dangling pointer)，会造成指针指向程序不想访问的位置。Java 使用自动的内存分配和无用单元回收机制来防止类似的错误发生。

图 2-14 总结了 MIPS 汇编语言的寄存器约定。这种约定是加速大概率事件的另外一个例子：传递 4 个参数、2 个寄存器用于返回结果、保存 8 个寄存器、10 个暂存器对于大多数过程调用来说足够使用。

名称	寄存器号	用途	调用时是否保存
\$zero	0	常数 0	不适用
\$v0 ~ \$v1	2 ~ 3	计算结果和表达式求值	否
\$a0 ~ \$a3	4 ~ 7	参数	否
\$t0 ~ \$t7	8 ~ 15	临时变量	否
\$s0 ~ \$s7	16 ~ 23	保存的寄存器	是
\$t8 ~ \$t9	24 ~ 25	更多临时变量	否
\$gp	28	全局指针	是
\$sp	29	栈指针	是
\$fp	30	帧指针	是
\$ra	31	返回地址	是

图 2-14 MIPS 寄存器约定。称为 \$at 的寄存器 1 被汇编器所保留（见 2.12 节），称为 \$k0 ~ \$k1 的寄存器 26 ~ 27 被操作系统所保留。关于这点也可见 MIPS 参考数据卡的第 2 列

01 精解 如果参数多于 4 个该怎么办呢？MIPS 约定将额外的参数放在栈中帧指针的上方。这样，过程从寄存器 \$a0 到 \$a3 中获得前 4 个参数，通过帧指针在内存中寻址获得其余参数。

如图 2-12 中所述，帧指针的方便性在于对过程中所有栈内的变量引用都具有相同的偏移。然而，帧指针并不是必需的。GNU MIPS C 编译器使用帧指针，而来自 MIPS 的 C 编译器则没有使用，它将寄存器 30 用作另一个保存的寄存器（\$s8）。

01 精解 一些递归过程可以不使用递归而用迭代的方式实现。通过消除过程调用的相关开销，迭代可以显著提高性能。例如，考虑下面一个用来求和的过程：

```
int sum (int n, int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}
```

考虑过程调用 sum(3, 0)。这将递归调用 sum(2, 3)、sum(1, 5) 和 sum(0, 6)，然后结果 6 将进行 4 次返回操作。这种求和的递归调用称为尾调用 (tail call)，而这个例子可以使用尾迭代 (tail recursion) 高效地实现（假设 \$a0 = n 且 \$a1 = acc）：

```
sum: slti $t0, $a0, 1          # test if n <= 0
     bne $t0, $zero, sum_exit # go to sum_exit if n <= 0
     add $a1, $a1, $a0         # add n to acc
     addi $a0, $a0, -1         # subtract 1 from n
     j sum                      # go to sum
sum_exit:
     add $v0, $a1, $zero       # return value acc
     jr $ra                     # return to caller
```

105

01 小测验

下面关于 C 和 Java 的描述哪些是正确的？

1. C 程序员显式地管理数据，而在 Java 中一般是自动的。
2. C 比 Java 导致更多的指针错误和内存泄漏错误。

2.9 人机交互

! (@ | = > (wow open tab at bar is great)

——键盘诗《Hatless Atlas》的第4行，1991

(对ASCII字符的一些命名：“!”是wow，“(”是open，“|”是bar，等等)

发明计算机是为了数字计算，不过计算机很快被用于商业方面的文字处理。今天大多数计算机使用8位的字节来表示字符，也就是几乎每个人都遵循的ASCII（American Standard Code for Information Interchange）码。图2-15对ASCII进行了总结。

ASCII值	字符	ASCII值	字符	ASCII值	字符	ASCII值	字符	ASCII值	字符	ASCII值	字符
32	space	48	0	64	@	80	P	96	`	112	p
33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	\$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	,	55	7	71	G	87	W	103	g	119	w
40	(56	8	72	H	88	X	104	h	120	x
41)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	Z	106	j	122	z
43	+	59	;	75	K	91	[107	k	123	{
44	,	60	<	76	L	92	\	108	l	124	
45	-	61	=	77	M	93]	109	m	125	_
46	.	62	>	78	N	94	^	110	n	126	~
47	/	63	?	79	O	95	-	111	o	127	DEL

图2-15 字符的ASCII码表示。注意所有大写字母和对应小写字母的差均为32，这个观测结果可以得到一条检查和切换大小写的简便方法。没有给出的ASCII值包括格式化字符。例如，8代表退格，9代表tab字符，而13代表回车。另外一个有用的值0表示null，C编程语言用这个来标记字符串的结尾。这些内容可以在MIPS参考数据卡的第3列中找到

106

01 例题·ASCII码与二进制数对比

我们可以使用一串ASCII码而不用整数来表示数字。如果用ASCII码表示10亿这个数将比用32位整数表示增加多少存储呢？

01 答案

10亿就是1 000 000 000，需要使用10位ASCII码表示，每一个ASCII码都是8位长。所以存储将增长到 $(10 \times 8) / 32$ ，即2.5倍。除了存储空间要增加外，用于对这些十进制数字进行加法、减法、乘法和除法的硬件的设计也是困难的。这些困难解释了为什么计算专家越来越相信使用二进制的计算机是自然的，而偶然出现的十进制计算机则是奇怪的。□

可以使用一系列指令从一个字中提取出一个字节，所以字的读取和存储同样可以完成对字节的传输。然而，由于在某些程序中对文本的操作十分普遍，所以MIPS还提供字节传输指令。字节读取1b（load byte）指令从内存中读出一个字节，并将其放在一个寄存器最右边的8位。

字节存储 sb (store byte) 指令把一个寄存器最右边的 8 位取出来然后写到内存中。这样，我们可以按下面的顺序复制一个字节：

```
lb $t0,0($sp)      # Read byte from source
sb $t0,0($gp)      # Write byte to destination
```

字符串通常被组合为字符数目可变的字符串。表示一个字符串的方式有三种选择：1) 保留字符串的第一个位置用于给出字符串的长度；2) 附加一个带有字符串长度的变量（如在结构体中）；3) 字符串最后的位置用一个字符来标识其结尾。C 语言使用第三种选择，用一个值为 0 (ASCII 码中的 null) 的字节来结束字符串。所以，字符串“Cal”在 C 中用 4 字节表示，用十进制表示分别为：67、97、108、0。（下面即将看到，Java 采用第一种表示方法。）

107

01 例题·通过编译一个字符串复制过程，来展示如何使用 C 字符串

strcpy 过程将 C 语言中约定使用 null 字节结束的字符串 y 复制到字符串 x：

```
void strcpy (char x[], char y[])
{
    int i;

    i = 0;
    while ((x[i] = y[i]) != '\0') /* copy & test byte */
        i += 1;
}
```

编译后的 MIPS 汇编代码是什么？

01 答案

下面是基本的 MIPS 汇编代码段。假定数组 x 和 y 的基地址在 \$a0 和 \$a1 中，而 i 在 \$s0 中。strcpy 调整栈指针然后将保存的寄存器 \$s0 保存在栈中。

```
strcpy:
    addi   $sp,$sp,-4  # adjust stack for 1 more item
    sw    $s0, 0($sp)  # save $s0
```

为了将 i 初始化为 0，下一条指令通过对 0 和 0 做加法并将和放到 \$s0 中的方法将 \$s0 置为 0：

```
add   $s0,$zero,$zero # i = 0 + 0
```

这是循环的开始。y [i] 地址的形成是通过把 i 加到 y [] 上：

```
L1: add   $t1,$s0,$a1 # address of y[i] in $t1
```

注意我们不必将 i 乘以 4，因为 y 是字节的数组而非字的数组，和前面的例子一样。

为了读取 y [i] 中的字符，我们使用无符号字节读取指令，将字符放入 \$t2 中：

```
lbu   $t2, 0($t1) # $t2 = y[i]
```

采用类似的计算方式将 x [i] 的地址放在 \$t3 中，然后将 \$t2 中的字符保存到该地址中。

```
add   $t3,$s0,$a0  # address of x[i] in $t3
sb   $t2, 0($t3)  # x[i] = y[i]
```

接下来，如果字符是 0 则退出循环。也就是说，如果它是字符串的最后一个字符则退出：

```
beq   $t2,$zero,L2 # if y[i] == 0, go to L2
```

如果不是，将 i 加 1 继续循环：

```
addi  $s0, $s0,1   # i = i + 1
j     L1           # go to L1
```

如果不继续循环，那就是到了字符串的最后一个字符，我们还原 \$s0 和栈指针，然后返回。

108

```

L2: lw      $s0, 0($sp) # y[i] == 0: end of string.
          # Restore old $s0
addi   $sp,$sp,4    # pop 1 word off stack
jr     $ra           # return

```

在 C 中字符串复制通常使用指针而不是数组，从而避免上面代码中对 *i* 的操作。详见 2.14 节数组和指针对比的相关解释。□

由于 *strcpy* 是一个叶过程，编译器可以把 *i* 放在临时寄存器中以避免对 *\$s0* 进行保存和恢复。因此，我们可以不把 *\$t* 寄存器用做临时寄存器，而是将其用作被调用者可以方便使用的寄存器。当编译器遇到一个叶过程时，它会在用完所有临时寄存器之后，才使用那些必须保存的寄存器。

Java 中的字符和字符串

Unicode 是大多数人类语言中字母的通用编码。图 2-16 是一个 Unicode 字母表的示例，Unicode 中字母数和 ASCII 编码中有用的字符数一样多。为了更有包容性，Java 对字符使用 Unicode，它默认使用 16 位来表示一个字符。
109

Latin	Malayalam	Tagbanwa	General Punctuation
Greek	Sinhala	Khmer	Spacing Modifier Letters
Cyrillic	Thai	Mongolian	Currency Symbols
Armenian	Lao	Limbu	Combining Diacritical Marks
Hebrew	Tibetan	Tai Le	Combining Marks for Symbols
Arabic	Myanmar	Kangxi Radicals	Superscripts and Subscripts
Syriac	Georgian	Hiragana	Number Forms
Thaana	Hangul Jamo	Katakana	Mathematical Operators
Devanagari	Ethiopic	Bopomofo	Mathematical Alphanumeric Symbols
Bengali	Cherokee	Kanbun	Braille Patterns
Gurmukhi	Unified Canadian Aboriginal Syllabic	Shavian	Optical Character Recognition
Gujarati	Ogham	Osmanya	Byzantine Musical Symbols
Oriya	Runic	Cypriot Syllabary	Musical Symbols
Tamil	Tagalog	Tai Xuan Jing Symbols	Arrows
Telugu	Hanunoo	Yijing Hexagram Symbols	Box Drawing
Kannada	Buhid	Aegean Numbers	Geometric Shapes

图 2-16 Unicode 字母表示例。Unicode 4.0 版本有超过 160 个“块”，每个块是一个符号集的名字，且是 16 的整数倍。例如，希腊字符（Greek）从 0370₁₆ 开始，西里尔字符（Cyrillic）从 0400₁₆ 开始。前三列以 Unicode 的数字顺序粗略地列出了 48 个块对应的 48 种人类语言。最后一列中的 16 个块是多种语言，并没有按照顺序排列。默认的是 16 位编码，称为 UTF-16。一种称为 UTF-8 的变长编码，将 ASCII 子集保持为 8 位，其余字符用 16 或 32 位来表示。UTF-32 使用 32 位表示一个字符。更多内容请参见 www.unicode.org

MIPS 指令集包含显式的读取和存储 16 位半字（halfword）的指令。读取半字指令 *lh*（load half）从存储器中读出一个半字，然后将其放在寄存器的最右边 16 位。与读取字节类似，读取半字指令 *lh* 也将半字看作有符号数并进行符号扩展，以填充寄存器左侧的 16 位。而无符号读取半字指令 *lhu*（load halfword unsigned）将半字看作无符号数，与 *lh* 相比，这条指令更加常用。存储半字指令 *sh*（store half）将寄存器最右边的 16 位写入存储器。我们按照下面的序列来复制半字：

```

lhu $t0,0($sp) # Read halfword (16 bits) from source
sh $t0,0($gp)  # Write halfword (16 bits) to destination

```

字符串是一个标准的 Java 类，它对连接、比较、转换的方法提供了专门的内建支持和预定方法。与 C 不同的是，Java 包含一个字来给出字符串长度，这和 Java 数组相似。

110

01 精解 MIPS 软件试图保持栈和字地址的对齐，这样就允许程序总是使用 `lw` 和 `sw`（要求必须是对齐的）来访问栈。这一约定意味着一个 `char` 类型变量在栈中被分配 4 字节，尽管它并不需要这么多。然而，一个 C 字符串变量或一个字节数组会把每 4 字节压缩为 1 个字，而一个 Java 字符串变量或 `short` 类型数组会把每 2 个半字压缩为 1 个字。

01 精解 为了反映 web 的全球性特征，当今的大部分 web 页面采用 Unicode，而非 ASCII。

01 小测验

I. 下面关于 C 和 Java 中字符和字符串的陈述哪些是正确的？

1. C 中一个字符串占用的内存是 Java 中同样字符串的一半。
2. C 和 Java 中字符串只是一个一维字符数组的非正规名字。
3. C 和 Java 中采用 `null` (0) 来标记字符串的结尾。
4. 对字符串的操作，例如求长度，在 C 中比在 Java 中更快。

II. 下面哪种类型的变量存放 $1\ 000\ 000\ 000_{10}$ 占用的内存空间最大？

1. C 语言的 `int`
2. C 语言的 `string`
3. Java 的 `string`

2.10 MIPS 中 32 位立即数和寻址

虽然保持所有 MIPS 指令为 32 位长简化了硬件，但有时使用 32 位常量或 32 位地址更加方便。本节先介绍使用较大常量的一般解决方法，然后描述了用于分支和跳转指令寻址的优化措施。

111

2.10.1 32 位立即数

尽管常数往往比较短而且适于 16 位字段，但有时它们会更大。MIPS 指令集中的读取立即数高位指令 `lui` (load upper immediate) 专门用于设置寄存器中常数的高 16 位，允许后续指令设置常数的低 16 位。图 2-17 描述了 `lui` 的操作。

指令 `lui $t0, 255#$t0 is register 8` 对应的机器语言：

001111	00000	01000	0000000011111111
<hr/>			
执行 <code>lui \$t0, 255</code> 后寄存器 <code>\$t0</code> 的值：			
0000000011111111	0000000000000000		

图 2-17 `lui` 指令的效果。`lui` 指令将 16 位立即数常量存放到寄存器的高 16 位，低 16 位用 0 填充

01 例题·加载 32 位常量

加载下面这个 32 位常量到寄存器 `$s0` 的 MIPS 汇编代码是什么？

0000 0000 0011 1101 0000 1001 0000 0000

01 答案

首先，我们使用命令 `lui` 加载高 16 位，十进制表示是 61：

`lui $s0, 61 # 61 decimal = 0000 0000 0011 1101 binary`

执行上面的指令后，寄存器 \$s0 的值为

0000 0000 0011 1101 0000 0000 0000 0000

下一步是插入低 16 位，十进制表示是 2304：

ori \$s0, \$s0, 2304 # 2304 decimal = 0000 1001 0000 0000

寄存器 \$s0 中的最终值就是所需要的值：

0000 0000 0011 1101 0000 1001 0000 0000

112

□

01 硬件/软件接口 编译器或汇编程序必须把大的常数分解为若干小的常数，然后再合并到一个寄存器中。正如你想象的那样，立即数字段大小的限制，无论在取/存数指令中对存储器的地址还是在立即数指令中对常数都可能带来问题。如果这项工作由汇编程序来做，如 MIPS 软件，那么汇编程序必须有一个可用的临时寄存器来创建长整数值。这是给汇编程序保留 \$at 寄存器的一个原因。

因此，MIPS 机器语言的符号表示不再受到硬件限制，但仍受汇编程序构造者所选择包括的内容的限制（见 2.12 节）。我们以靠近硬件层的方式解释计算机的体系结构，需要注意的是，我们所使用汇编程序的增强扩展语言，在实际处理器中是不存在的。

01 精解 构造 32 位常数时必须小心。指令 addi 将指令最左边的 16 位立即数字段复制到一个字的高 16 位中。2.6 节的立即数逻辑或操作（logical or immediate）把 0 读到高 16 位中，所以可被汇编程序用于和 lui 一起创建 32 位常数。

2.10.2 分支和跳转中的寻址

MIPS 跳转指令寻址采用最简单的寻址方式。它们使用最后一种 MIPS 指令格式，称为 J 型。J 型除了 6 位操作码之外，其余位都是地址字段。所以

j 10000 # go to location 10000

可以汇编为下面的格式（实际中要更加复杂一些，正如我们后面将看到的那样）：

2	10000	
6位		26位

其中跳转操作码的值为 2，跳转地址为 10000。

和跳转指令不同，条件分支指令除了规定分支地址之外还必须指定两个操作数。因此

bne \$s0,\$s1,Exit # go to Exit if \$s0 ≠ \$s1

被汇编为下面的指令，只保留了 16 位用于指定分支地址：

5	16	17	Exit
6位	5位	5位	16位

如果让程序地址适应该 16 位字段，则意味着任何程序都不能大于 2^{16} ，这在今天来说太小，因此是一种很不现实的选择。另一个可选的办法是指定一个总是加到分支地址上的寄存器，这样分支指令的地址可按如下方式计算：

$$\text{程序计数器} = \text{寄存器} + \text{分支地址}$$

这个求和结果允许程序的大小达到 2^{32} ，并且仍能使用条件分支，从而解决了分支地址大小的问题。随之而来的问题是使用哪个寄存器呢？

答案取决于条件分支是如何使用的。条件分支在循环和 if 语句中都可以找到，它们倾向于转到附近的指令。例如，在 SPEC 基准测试程序中，大概一半条件分支的跳转距离小于 16 条指令。因为程序计数器（Program Counter, PC）包含当前指令的地址，如果我们使用 PC 来作为

113

增加地址的寄存器，我们可转移到离当前指令距离为 $\pm 2^{15}$ 个字的地方。几乎所有循环和if语句都远远小于 2^{16} 个字，因此PC是一个理想的选择。

这种分支寻址形式称为PC相对寻址(PC-relative addressing)。正如在第4章中将会看到的那样，提前递增PC来指向下一条指令会对硬件带来很多方便。所以，MIPS寻址实际上是相对于下一条指令的地址(PC+4)，而不是相对于当前指令(PC)。寻址附近的指令是加速大概率事件的另外一个例子。

② PC相对寻址：一种寻址方式，它将PC和指令中的常数相加作为寻址结果。

像近期的大多数计算机一样，MIPS对所有条件分支使用PC相对寻址，因为这些指令的跳转目标一般都比较接近其分支地址。另一方面，跳转链接指令并非总是靠近调用者的过程，所以它们通常使用其他寻址方式。因此，MIPS体系结构通过使用跳转和跳转链接指令的J型格式来为过程调用提供长地址。

因为所有MIPS指令都是4字节长，所以在PC相对寻址时所加的地址被设计为字地址而不是字节地址。相对于16位的字节地址，16位的字地址跳转范围扩大了4倍。同样，跳转指令的26位字段也是字地址，它可以表示28位的字节地址。

01 精解 因为PC是32位，所以有4位必须来自于跳转指令之外的其他地方。MIPS跳转指令仅仅代替PC的低28位，而高4位保持不变。装载器和链接器(见2.12节)必须十分小心以避免程序超过256MB的寻址界限(6400万条指令)；否则，该跳转必须替换为寄存器跳转指令，并在执行前使用其他指令将完整的32位地址加载到一个寄存器中。

114

01 例题·在机器语言中描述分支偏移

假设2.7.1节的while循环语句被编译成下面的MIPS汇编代码：

```
Loop: sll $t1,$s3,2      # Temp reg $t1 = 4 * i
    add $t1,$t1,$s6      # $t1 = address of save[i]
    lw   $t0,0($t1)       # Temp reg $t0 = save[i]
    bne $t0,$s5, Exit    # go to Exit if save[i] ≠ k
    addi $s3,$s3,1        # i = i + 1
    j    Loop             # go to Loop
Exit:
```

如果我们假设把loop的开始位置放在内存的80 000处，那么该循环的MIPS机器代码是什么呢？

01 答案

汇编指令和它们的地址如下：

80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8		0	
80012	5	8	21		2	
80016	8	19	19		1	
80020	2			20000		
80024	...					

注意MIPS指令使用字节寻址，所以相邻字的地址相差4，即一个字中的字节的数量。第4行的bne指令将2个字或是8字节加到下一条指令地址(80016)上，使用相对下一条指令的偏移($8 + 80016$)指明跳转目标，而不是使用相对该分支指令的偏移($12 + 80012$)，也不是使用完整的目的地址(80024)。最后一行的跳转指令采用完整的地址($20000 \times 4 = 80000$)，

115 对应于 Loop 标签。 □

01 硬件/软件接口 大多数条件分支都转移到一个附近的位置，但有时也会转移很远，距离超过条件分支指令的 16 位可以表示的范围。汇编器的解决方法就像处理对大地址或大常数的方法一样：插入一个跳转到分支目标的无条件跳转，并将条件取反以便由分支决定是否跳过该无条件跳转指令。

01 例题·远距离的分支转移

假设在寄存器 \$s0 与寄存器 \$s1 值相等时需要跳转，可以使用如下指令：

beq \$s0, \$s1, L1

用两条指令替换上面的指令，以获得更远的转移距离。

01 答案

可用下面的指令替换短地址的条件分支指令：

```
bne $s0, $s1, L2
j L1
L2:
```

□

2.10.3 MIPS 寻址模式总结

多种不同的寻址形式一般统称为寻址模式（addressing mode），图 2-18 给出了每种寻址模式的操作数如何识别。MIPS 寻址模式如下所示：

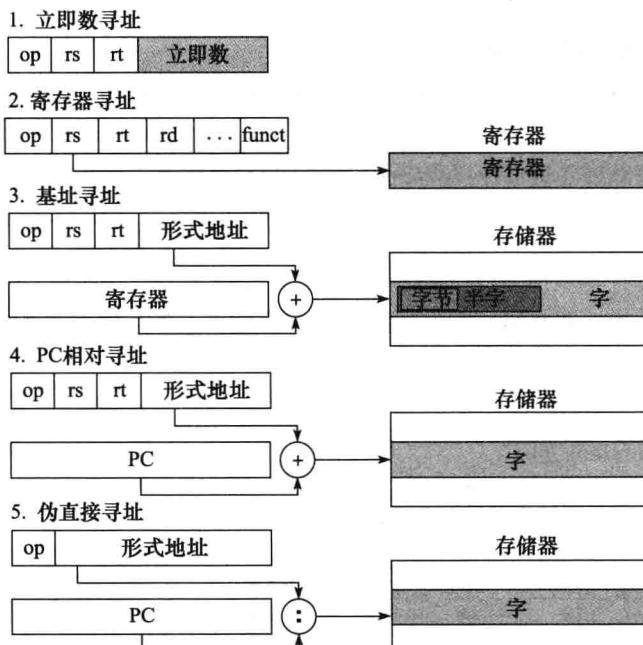


图 2-18 MIPS 5 种寻址模式的说明。阴影部分为操作数。模式 3 的操作数在内存中，而模式 2 的操作数是寄存器。注意，读数和存数对字节、半字或字有多种版本。模式 1 的操作数是指令自身的 16 位字段。模式 4 和模式 5 寻址的指令在内存中，模式 4 把 16 位地址左移 2 位与 PC 相加，而模式 5 把 26 位地址左移 2 位与 PC 计数器的高 4 位相连。注意，一种操作可能可以使用多种寻址模式，例如，加法可以使用立即数寻址 (addi) 和寄存器寻址 (add)

寻址模式：根据对操作数和/或地址的使用不同加以区分的多种寻址方式中的一种。

- 1) 立即数寻址 (immediate addressing)，操作数是位于指令自身中的常数。
- 2) 寄存器寻址 (register addressing)，操作数是寄存器。
- 3) 基址寻址 (base addressing) 或偏移寻址 (displacement addressing)，操作数在内存中，其地址是指令中基址寄存器和常数的和。
- 4) PC 相对寻址 (PC-relative addressing)，地址是 PC 和指令中常数的和。
- 5) 伪直接寻址 (pseudodirect addressing)，跳转地址由指令中 26 位字段和 PC 高位相连而成。

116

01 硬件/软件接口 虽然我们把 MIPS 系统结构按 32 位地址描述，但是几乎所有的微处理器（包括 MIPS）都能进行 64 位地址扩展（见附录 E 和 2.18 节）。这些扩展主要是针对大型程序的需要。指令集的扩展使得体系结构发展的同时，保持软件和下一代体系结构的向上兼容性。

117

2.10.4 机器语言解码

有时候必须通过逆向工程将机器语言恢复到最初的汇编语言，比如检查“核心转储”(core dump)时。图 2-19 描述了 MIPS 机器语言对各个字段的编码。该图可用于汇编语言和机器语言之间的手动翻译。

01 例题·机器码解码

下面这条机器指令对应的汇编语言语句是什么？

00af8020hex

01 答案

第一步是将十六进制转换到二进制，以便找到操作码字段：

(Bits: 31 28 26)	5 2 0)
0000 0000 1010 1111 1000 0000 0010 0000	

我们查看操作码字段来决定指令的操作类型。参照图 2-19，当 31~29 位是 000 且 28~26 位也是 000 时，它是 R 型指令。参照图 2-20，将该二进制指令按照 R 型指令字段重新排列：

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

op(31:26)

28~26 31~29	0(000)	1(101)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R 型	Bltz/gez	跳转	跳转并链接	分支 eq	分支 ne	blez	bgtz
1(001)	立即数加法	addiu	小于立即数置位	小于无符号立即数时置位	andi	ori	xori	取立即数高位
2(010)	TLB	FIPt						

图 2-19 MIPS 指令解码。该标记根据行和列确定字段的值。例如，图的顶部在第 4 行（指令的第 31~29 位为 100_2 ）第 3 列（指令的第 28~26 位为 011_2 ）描述了取字指令，因此相应操作码字段（第 31~26 位）的 (R 型) 值是 100011_2 。下划线表示该字段在其他地方被使用。例如，第 0 行第 0 列 ($op = 000000_2$) 的 R 型在图的底部定义。因此，底部第 4 行第 2 列的 subtract 意味着指令 funct 字段（第 5~0 位）是 100010_2 而操作码字段（第 31~26 位）是 000000_2 。第 2 行第 1 列的 FIPt 在第三章的图 3-18 中定义。Bltz/gez 是附录 B 中 4 条指令的操作码：bltz、bgez、bltzal 和 bgezal。附录 A 涵盖所有的指令

op(31:26)								
3(011)								
4(100)	取字节	取半字	lw1	取字	取无符号字节	取无符号半字	lw1	
5(101)	存字节	存半字	sw1	存字			sw1	
6(110)	取链接字	lwc1						
7(111)	存条件字	swc1						

op(31:26) = 010000(TLB), rs(25:21)								
23 ~ 21 25 ~ 24	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(00)	mfc0		cfc0		mtc0		ctc0	
1(01)								
2(10)								
3(11)								

op(31:26) = 000000(R型), funct(5:0)								
2 ~ 0 5 ~ 3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	逻辑左移		逻辑右移	sra	sllv		srlv	sraw
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or(nor)
5(101)			set l.t.	set l.t. 无符号操作				
6(110)								
7(111)								

图 2-19 (续)

名称	字段						备注
字段大小	6 位	5 位	5 位	5 位	5 位	6 位	所有 MIPS 指令都是 32 位
R 型	op	rs	rt	rd	shamt	funct	算术指令型
I 型	op	rs	rt	地址/立即数			传输、分支和立即数型
J 型	op	目标地址					跳转指令型

图 2-20 MIPS 指令的格式

图 2-19 的底部确定了 R 型指令的操作。在本例中，5~3 位是 100，2~0 位是 000，因此该二进制指令为 add 指令。

下面我们通过查找字段值来解码指令的剩余部分。rs 字段的十进制值是 5，rt 是 15，rd 是 16（shamt 未使用）。图 2-14 说明这些数字表示寄存器 \$a1、\$t7 和 \$s0。现在可以给出转换后的汇编指令：

```
add $s0,$a1,$t7
```

图 2-20 给出了所有 MIPS 指令的格式。第 2.2 节的图 2-1 汇总了本章出现的所有汇编指令。其他 MIPS 指令主要处理算术运算和实数，将在第 3 章介绍。□

01 小测验

- I. 在 MIPS 中条件分支的地址范围 ($K = 1024$) 是多大?
 1. 地址在 $0 \sim 64K - 1$ 之间
 2. 地址在 $0 \sim 256K - 1$ 之间
 3. 分支前后地址范围各大约 $32K$
 4. 分支前后地址范围各大约 $128K$

- II. 在 MIPS 中跳转和跳转链接指令的地址范围 ($M = 1024K$) 是多大?
 1. 地址在 $0 \sim 64M - 1$ 之间
 2. 地址在 $0 \sim 256M - 1$ 之间
 3. 分支前后地址范围各大约 $32M$
 4. 分支前后地址范围各大约 $128M$
 5. 由 PC 提供高 6 位地址的 $64M$ 大小的块中任意地址
 6. 由 PC 提供高 4 位地址的 $256M$ 大小的块中任意地址

- III. 机器指令 $0000\ 0000_{16}$ 对应的 MIPS 汇编语言指令是什么?
 1. j
 2. R 型
 3. addi
 4. sll
 5. mfc0
 6. 未定义的操作码：没有对应 0 的合法指令

118

l

120

2.11 并行与指令：同步

当任务之间相互独立的时候，任务的并行执行是比较容易的。但往往任务之间需要相互协作，这种协作通常意味着某些任务写的结果是其他任务需要读取的值。这时执行读任务的一方要知道写任务什么时候完成了写操作，才能安全地读回数据。就是说，任务之间需要同步 (synchronize)，否则就有发生数据竞争 (data race) 的危险，导致读数据错误而引起程序运行结果的改变。

数据竞争：假如来自不同线程的两个访存请求访问同一个地址，它们连续出现，并且至少其中一个是一个写操作，那么这两个存储访问形成数据竞争。

例如，回忆第 1 章 1.8 节所提到的 8 个记者共同写作一个故事的例子。假设一个记者要写总结，他要阅读所有之前的章节。因此，他必须知道其他记者什么时候可以完成各自的章节，然后他再撰写总结，这样他就不用担心写好总结后其他记者再对各自章节进行修改。所以，他们就需要很好地同步各个章节撰写和阅读的过程，这样总结才能和前面章节中所写的内容相一致。

在计算中，同步机制要依赖硬件提供的同步指令，这些指令可由用户调用。本节我们重点讨论加锁 (lock) 和解锁 (unlock) 同步操作的实现。采用加锁和解锁可以直接创立一个仅允许单个处理器操作的区域，叫作互斥 (mutual exclusion) 区。更复杂的同步机制实现也与此类似。

在多处理器中实现同步需要一组硬件原语，提供对存储单元进行原子读和原子写的能力，使得在进行存储器原子读或原子写操作时任何其他操作都不得插入。如果没有这样的硬件原语，那么建立同步机制的代价将会变得很高，并且随着处理器数量的增加情况将更为恶化。

建立基本硬件原语有若干可选的方案，这些方案都可以实现原子读和原子写的功能，并能

用某种方法表示这些操作是否为原子操作。通常，体系结构设计人员并不希望基本硬件原语被用户使用，而是希望这些原语被系统程序员用来建立同步库，建立同步库的过程通常很复杂且难度较大。

我们用原子交换原语（atomic exchange 或 atomic swap）来演示如何建立基本同步机制。这个原语是将寄存器中的一个值和存储器中的一个值相互交换。

为了展示该原语建立同步原理的基本过程，假定使用存储器中某个单元来表示一个锁变量：其数值为 0 时表示解锁，为 1 时表示加锁。一个处理器尝试对锁单元加锁，方法是用一个寄存器中的 1 与该锁单元的值进行交换。交换以后该锁单元的新值为 1，返回值（锁单元的原值）如果是 1，表明这个锁已被其他处理器占用；否则返回值为 0，表示锁是自由的，尝试加锁成功。此时锁单元已被修改成 1，以防止任何其他处理器再来占用。
121

例如，考虑有两个处理器同时尝试进行交换操作，它们的竞争关系就会被破坏。因为其中只能有一个处理器先执行交换操作，并且返回 0。那么第二个处理器执行完交换操作的时候返回值就变成了 1。用交换原语实现同步的关键是操作的原子性：交换操作是不可分割的，并且由硬件对两个同时执行的交换操作进行排序。有可能两个处理器同时尝试置位同步变量，但这两个处理器认为它们同时成功设置了同步变量是不可能的。

实现单个的原子存储器操作给处理器的设计者带来了若干挑战，因为这要求存储器的读、写操作都是有单条不可被中断的指令完成。

一种可行的方法是采用指令对，其中第二条指令返回一个表明这对指令是否原子执行的标志值。假如处理器的操作都是在这对指令之前或之后执行，这对指令就是原子的。因此，当一个指令对是原子的，没有哪个处理器能改变这两个指令执行之间的数据值。

在 MIPS 处理器中这一指令对包括一条叫作链接取数（load linked）的特殊取数指令和一条叫作条件存数（store conditional）的特殊存数指令。我们顺序地使用这两条指令：如果链接取数指令所指定的锁单元的内容在相同地址的条件存数指令执行前已被改变，那么条件存数指令就执行失败。我们定义条件存数指令完成以下功能：保存寄存器的值，并且如果执行成功则将寄存器的值修改为 1，如果失败则修改为 0。因为链接取数指令返回锁单元的原始值，条件存数指令执行成功的时候才返回 1，下面的指令序列实现了存储器单元的原子交换。存储器单元的地址由 \$s1 中的值指出。

```
again: addi $t0,$zero,1      ;copy locked value
      li    $t1,0($s1)       ;load linked
      sc    $t0,0($s1)       ;store conditional
      beq   $t0,$zero,again  ;branch if store fails
      add   $s4,$zero,$t1     ;put load value in $s4
```

在 `li` 和 `sc` 两条指令之间的任何时候有处理器插入，并修改了该锁单元的值，指令 `sc` 都会将 `$t0` 置为 0，引起这段指令序列重新执行。在指令序列的最后，寄存器 `$s4` 中的值和 `$s1` 指向的锁单元的值发生了原子交换。

01 精解 尽管我们讲述的同步是在多处理器系统中的，但是原子操作对于单个处理器上运行的操作系统在处理多个进程时也是十分有用的。在单处理器中，为了保证执行不被任何事件所干扰，条件存数指令在处理器两条指令之间进行上下文切换（context switch）时也会失败（见第 5 章）。
122

链接取数/条件存数机制的优点是：可以通过它们来构造其他的诸如原子比较和交换（atomic compare and swap）或者原子取后加（atomic fetch-and-increment）等同步原语。这些同步原语可以被用在一些并行编程模型中。这些同步原语的实现需要在 `li` 指令和 `sc` 指令之间插入更多的指令，但不需要太多。

因为在链接取数指令执行之后，任何试图修改锁单元值的操作或者任何异常都将导致条件存数指令执行失败，所以在选择 `ii` 和 `sc` 之间的指令时就要格外注意。特别需要注意的是，允许使用的并且不会造成问题的只有寄存器 - 寄存器指令，而处理器可能由于重复的页错误而导致始终无法完成 `sc` 指令，从而使处理器处于一种死锁的状态。另外，链接取数和条件存数之间的指令数一定要尽可能少，这样才可以减少不相关的事件或者竞争资源的处理器所引起条件存数指令执行失败的频率。

01 小测验

什么时候才会用到像链接取数（load linked）和条件存数（store conditional）这样的原语？

- 当一个并行程序中相互协作的线程需要同步以获得对共享数据的正确的读写行为时
- 当运行在单处理器上的相互协作的处理过程需要同步以获得对共享数据的正确的读写行为时

2.12 翻译并执行程序

本节描述了将存储在硬盘文件中的 C 程序转换为可执行程序的 4 个步骤，图 2-21 所示是语言翻译的层次。尽管某些系统可能合并部分步骤以减少转换时间，但从逻辑上讲，这 4 个步骤是程序转换流程所必经的 4 个阶段。本节将根据这种翻译层次进行描述。

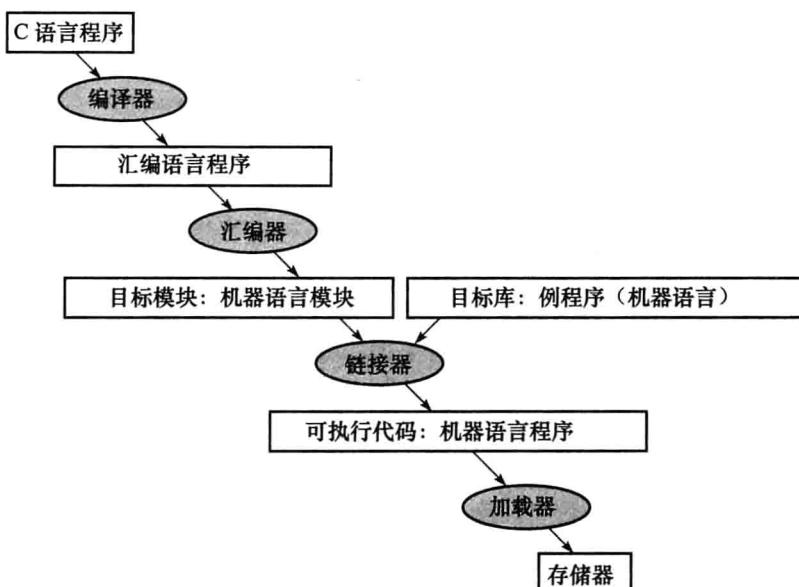


图 2-21 C 语言的翻译层次。用高级语言编写的程序首先需要被编译成为汇编语言，然后被汇编成机器语言组成的目标模块。链接器将多个模块和库程序组合在一起解析所有的引用。加载器将可执行程序加载到内存的适当位置，然后处理器就可以执行了。为了加快翻译的速度，某些步骤被跳过或和其他步骤组合在一起。一些编译器直接产生目标模块，一些系统使用带链接功能的加载器直接完成后面两步。为了确定文件的类型，UNIX 使用文件的后缀，`x.c` 代表 C 源文件，`x.s` 表示汇编文件，`x.o` 表示目标文件，`x.a` 表示静态链接库，`x.so` 表示动态链接库，默认情况下，`a.out` 表示可执行文件。MS-DOS 使用后缀 `.C`, `.ASM`, `.OBJ`, `.LIB`, `.DLL` 和 `.EXE` 来完成同样的功能。

2.12.1 编译器

编译器将 C 程序转换成一种机器能理解的符号形式的汇编语言程序（assembly language）

program)。高级语言编写的程序比使用汇编语言编写的代码少得多，所以程序员效率更高。

1975 年，因为存储器容量较小并且编译器效率不高，所以许多操作系统和汇编器都用汇编语言 (assembly language) 编写。如今单 DRAM 芯片容量增长 500 000 倍，减轻了人们对程序大小的关注，并且今天优化的编译器能够产生出几乎与一个汇编语言专家所写的程序一样好的汇编程序，对于大型程序有时甚至效果更好。

● 汇编语言：一种符号语言，能被翻译成二进制的机器语言。

2.12.2 汇编器

因为汇编语言对于高层次软件是一个接口，所以汇编器也能够处理一些机器语言指令的常见变种，就像这些变种是它自己的指令一样。硬件不需要实现这些指令，然而它们在汇编语言中的存在简化了程序转换和编程。这类指令称为伪指令 (pseudoinstruction)。

● 伪指令：汇编语言指令的一个变种，通常被看作一条汇编指令。

如前所述，MIPS 硬件确保寄存器 \$zero 保持 0 值。即一旦使用寄存器 \$zero，它都提供 0，而且程序员不能修改寄存器 \$zero 的值。寄存器 \$zero 用于生成汇编语言指令 move，move 的功能是将一个寄存器中的内容复制到另一个中。因此即使 MIPS 体系结构中不存在这条指令，MIPS 汇编器也能够识别它：

```
move $t0,$t1      # register $t0 gets register $t1
```

汇编器将这条汇编语言指令转换成功能等价的如下机器语言指令：

```
add $t0,$zero,$t1 # register $t0 gets 0 + register $t1
```

在 2.7.1 节的例子中提到，MIPS 汇编器将 blt (branch on less than，小于则分支) 转换成两条指令：slt 和 bne。其他例子包括 bgt、bge 和 ble。它也将一个到远距离的分支指令拆成一个分支指令和一个跳转指令。如前所述，MIPS 汇编器允许将 32 位常量加载到一个寄存器中，不用考虑立即数指令的 16 位限制。

总的来说，伪指令使 MIPS 拥有比硬件所实现的更为丰富的汇编语言指令集。唯一的代价是保留了一个由汇编器使用的寄存器 \$at。如果你打算写汇编程序，请使用伪指令来简化你的任务。为了理解 MIPS 体系结构并保证获得最好的性能，可以学习图 2-1 和图 2-19 中真正的 MIPS 指令。

汇编器同样接受不同基数的数字。除了二进制和十进制，它们通常还使用比二进制更为紧凑而又容易转化为位模式的基数。MIPS 汇编器使用十六进制。

这种特性相当方便，但是汇编器的主要任务是汇编成机器代码。汇编器将汇编语言程序转换成目标文件 (object file)，它包括机器语言指令、数据和指令正确放入内存所需要的信息。

为了产生汇编语言程序中每条指令对应的二进制表示，汇编器必须处理所有标号对应的地址。汇编器将分支和数据传输指令中用到的标号都放入一个符号表 (symbol table) 中。正如你所想的，这个表由标号和地址成对构成。

● 符号表：一个用来匹配标记名和指令所在内存字的地址的列表。

UNIX 系统中的目标文件通常包含以下 6 个不同的部分：

- 目标文件头，描述目标文件其他部分的大小和位置。
- 代码段，包含机器语言代码。
- 静态数据段，包含在程序生命周期内分配的数据。（UNIX 系统允许程序使用静态数据，它

存在于整个程序中；也允许使用动态数据，它随程序的需要而增长或缩小。见图 2-13。)

- 重定位信息，标记了一些在程序加载进内存时依赖于绝对地址的指令和数据。
- 符号表，包含未定义的剩余标记，如外部引用。
- 调试信息，包含一份说明目标模块如何编译的简明描述，这样，调试器能够将机器指令关联到 C 源文件，并使数据结构也变得可读。

125

下一小节描述了如何链接已经汇编完成的子程序，如库程序。

2.12.3 链接器

到目前为止我们所描述的内容表明，对于源程序任意一行代码的修改都需要重新编译和汇编整个程序。全部重新翻译是对计算资源的严重浪费。这种重复对于标准库程序尤为浪费，因为程序员要编译和汇编那些在定义上几乎从未改变过的过程。另一种方法是单独编译和汇编每个过程，以使得某一行代码的改变只需要编译和汇编一个过程。这种方法需要一个新的系统程序，称为链接编辑器或（link editor）链接器（linker），它把所有独立汇编的机器语言程序“拼接”在一起。

- 链接器：也称链接编辑器。它是一个系统程序，把各个独立汇编的机器语言程序组合起来并且解决所有未定义的标记，最后生成可执行文件。

链接器的工作分 3 个步骤：

- 1) 将代码和数据模块象征性地放入内存。
- 2) 决定数据和指令标签的地址。
- 3) 修补内部和外部引用。

链接器使用每个目标模块中的重定位信息和符号表，来解析所有未定义标签。这种引用发生在分支指令、跳转指令和数据寻址处，所以这个程序的工作非常像一个编辑器：它寻找所有旧地址并用新地址取代它们。编辑是“链接编辑器”或链接器名字的简称。采用链接器的原因是修补代码比重新编译和汇编要快得多。

如果所有外部引用都解析完，链接器接着决定每个模块将要占用的内存位置。回忆 2.8.4 节的图 2-13，它描述了 MIPS 在内存中为程序和数据分配空间的方式。因为文件是单独汇编的，所以汇编器不可能知道该模块的指令和数据相对于其他模块而言将会被放到哪里。当链接器将一个模块放到内存中的时候，所有绝对引用（absolute reference），即与寄存器无关的内存地址必须重定位以反映它的真实地址。

链接器产生一个可执行文件（executable file），它可以在一台计算机上运行。通常，这个文件与目标文件具有相同的格式，但是它不包含未解决的引用。具有部分链接的文件是可能的，如库程序，在目标文件中仍含有未解决的地址。

- 可执行文件：一个具有目标文件格式的功能程序，不包含未解决的引用。它可以包含符号表和调试信息。“剥离的可执行程序”不包含这些信息，可能包含加载器所需的重定位信息。

126

例题·目标文件的链接

将下面的两个目标文件链接。给出最终可执行文件中前几条指令对应的更新过的地址。为了便于理解，我们使用汇编语言来表示指令，在实际文件中，这些指令由数字表示。

注意目标文件中，我们已将必须在链接进程中更新的地址和标记高亮显示了，分别是引用过程 A 和过程 B 的地址的指令，以及引用数据字 X 和 Y 的地址的指令。

目标文件头			
	名字	过程 A	
	正文大小	100_{16}	
	数据大小	20_{16}	
代码段	地址	指令	
	0	<code>lw \$a0,0(\$gp)</code>	
	4	<code>jal 0</code>	
	
数据段	0	(X)	
	
重定位信息地址	地址	指令类型	依赖
	0	<code>lw</code>	X
	4	<code>jal</code>	B
符号表	标记	地址	
	X	—	
	B	—	
目标文件头			
	名字	过程 B	
	正文大小	200_{16}	
	数据大小	30_{16}	
代码段	地址	指令	
	0	<code>sw \$a1,0(\$gp)</code>	
	4	<code>jal 0</code>	
	
数据段	0	(Y)	
	
重定位信息地址	地址	指令类型	依赖
	0	<code>sw</code>	Y
	4	<code>jal</code>	A
符号表	标记	地址	
	Y	—	
	A	—	

127

01 答案

过程 A 需要找到 load 指令中标号为 X 的变量的地址和 jal 指令中过程 B 的地址。过程 B 需要找到 store 指令中标号为 Y 的变量的地址和 jal 指令中过程 A 的地址。

从 2.8.4 节的图 2-13 中，我们可以看到代码段从地址 $40\ 0000_{16}$ 开始而数据段从地址 $1000\ 0000_{16}$ 开始。过程 A 的正文被放置在第一个地址而它的数据被放置在第二个地址。过程 A 的目标文件头表明其代码段大小是 100_{16} 字节而数据段大小是 20_{16} 字节，这样过程 B 的代码段开始地址就是 $400\ 100_{16}$ ，数据段开始地址是 $1000\ 0020_{16}$ 。

可执行文件头			可执行文件头		
	正文大小	300_{16}		$0040\ 0104_{16}$	<code>jal 400\ 000_{16}</code>
	数据大小	50_{16}	
代码段	地址	指令	数据段	地址	
	$0040\ 0000_{16}$	<code>lw \$a0,8000_{16}(\$gp)</code>		$1000\ 0000_{16}$	(X)
	$0040\ 0004_{16}$	<code>jal 400100_{16}</code>	
		$1000\ 0020_{16}$	(Y)
	$0040\ 0100_{16}$	<code>sw \$a1,8020_{16}(\$gp)</code>	

图 2-13 也表明了代码段的起始地址是 $40\ 0000_{16}$ ，数据段的起始地址是 $1000\ 0000_{16}$ 。过程 A 的正文被放置在第一个地址而它的数据被放置在第二个地址。过程 A 的目标文件头表明其代码段大小是 100_{16} 字节而数据段大小是 20_{16} 字节，这样过程 B 的代码段开始地址就是 400100_{16} ，数据段开始地址是 10000020_{16} 。[⊖]

现在链接器更新了指令的地址字段。它使用指令类型字段得到待编辑地址的格式。这里共有两种类型：

1) jal 类型比较简单。因为它们使用伪直接寻址。对于地址 $40\ 0004_{16}$ 处的 jal，其地址字段是 $40\ 0100_{16}$ （程序 B 的地址），而地址 $40\ 0104_{16}$ 处的 jal 的地址字段是 $40\ 0000_{16}$ （程序 A 的地址）。

2) 存取数指令对应的地址更为复杂，因为它们和基址寄存器有关。本例使用全局指针作为基址寄存器。图 2-13 表明 \$gp 的初始值为 10008000_{16} 。为了得到地址 10000000_{16} （字 X 的地址），我们设置位于地址 400000_{16} 处的 lw 的地址字段中为 8000_{16} 。同样，为了得到地址 10000020_{16} （字 Y 的地址），可以设置位于地址 400100_{16} 处的 sw 的地址字段中为 8020_{16} 。 □

128

01 精解 回忆前面讲过 MIPS 指令是按字对齐的。所以 jal 指令丢弃最右侧 2 位来增加指令寻址范围。这样，它就可以使用 26 位来产生一个 28 位的字节地址。因此，本例中 jal 指令的低 26 位存放的实际地址是 $10\ 0040_{16}$ ，而不是 $40\ 0100_{16}$ 。

2. 12. 4 加载器

现在可执行文件已经在磁盘中，操作系统可以将其读入内存并启动执行它。在 UNIX 系统中，加载器（loader）按照如下步骤工作：

- 1) 读取可执行文件头来确定代码段和数据段的大小。
- 2) 为正文和数据创建一个足够大的地址空间。
- 3) 将可执行文件中的指令和数据复制到内存中。
- 4) 把主程序的参数（如果存在）复制到栈顶。
- 5) 初始化机器寄存器，将栈指针指向第一个空位置。
- 6) 跳转到启动例程，它将参数复制到参数寄存器并且调用程序的 main 函数。当 main 函数返回时，启动例程通过系统调用 exit 终止程序。

加载器：把目标程序装载到内存中以准备运行的系统程序。

附录 A 中的 A. 3 节和 A. 4 节更加详细地描述了链接器和加载器。

2. 12. 5 动态链接库

事实上，计算机科学中的每个问题可以在其他层次上间接地解决。

——David Wheeler

本节的第一部分将描述程序运行前链接库文件的传统方法。尽管这种静态的方法是最快的调用库程序的办法，但它有以下缺点：

- 库程序成为可执行代码的一部分。这样如果发布新版本的库以修正一些错误或支持新的硬件设备，静态链接的程序中使用的还是旧版本。
- 在程序运行时，尽管可能不会使用库中的所有部分，但它们还是会全部加载进来。相对程序而言，库可能会很大，例如，标准的 C 库有 2.5MB。

⊖ 原书中表格前后的文字除开头外一模一样。此处按原书翻译。——译者注

这些不足导致了动态链接库 (dynamically linked library, DLL) 的产生，也就是说，直到程序运行的时候，这些库例程才会被链接并加载。程序和库例程都会在非局部的过程和名字中保存额外的信息。在最初版本的 DLL 中，加载器调用一个动态链接器，使用文件中的额外信息来找到适当的库并且更新所有外部引用。

129 动态链接库：在程序执行过程中才被链接的库例程。

最初版本 DLL 的缺点是它仍链接库中所有在程序运行时可能调用的例程，而不是仅仅链接程序运行时实际调用的例程。由此产生 DLL 的晚过程链接 (lazy procedure linkage) 版本，该版本中每个例程只有在它被调用后才被链接。

就像这个领域中的许多创新一样，这个技巧采用了一种间接的方法。图 2-22 展示了该技术。它以一个非局部例程开始，该例程的末尾调用了一组虚例程，每个非局部例程都有一个人口。每个虚入口都包含一个间接跳转。

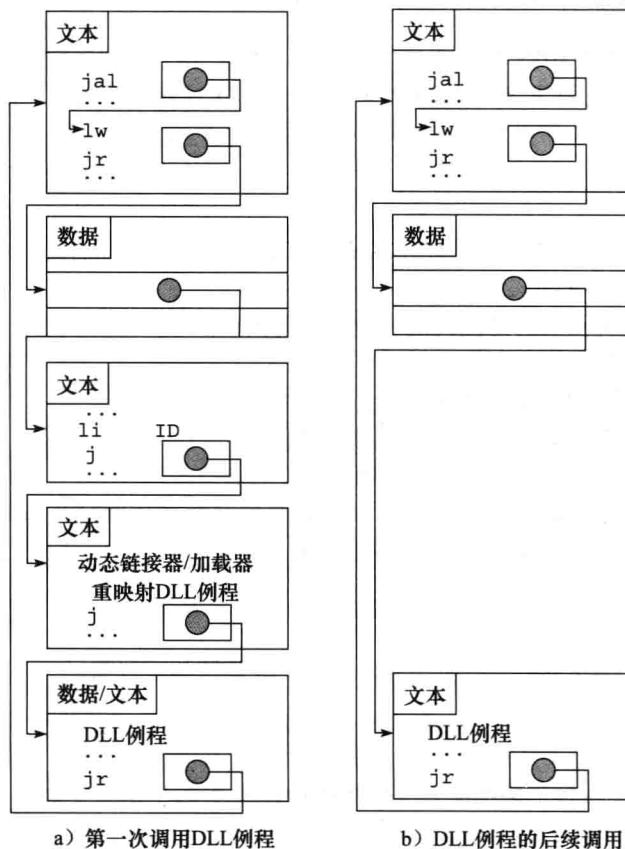


图 2-22 通过晚过程链接方式链接动态链接库。a) 第一次调用 DLL 的步骤；b) 在随后的调用中，查找例程、重映射例程和链接例程被跳过。我们将在第 5 章看到，操作系统通过虚拟内存管理方式来重映射例程以避免复制所需例程

第一次调用库例程的时候，程序首先调用虚入口，然后执行间接跳转。它通过将一个数字放入寄存器来识别所需的库例程，然后跳转到动态链接器或加载器。链接器或加载器找到所需的例程，将其重映射并改变间接跳转位置的地址，使其指向这个例程。然后跳转到这个例程。这个例程完成时，将返回到初始调用点。此后，它都会间接跳转到这个例程而不去执行额外的中间过程。

总的来说，DLL 需要额外的空间来存储动态链接的信息，但是不需要复制或链接整个库。仅仅在例程的第一次调用时开销较大，此后就只需一个间接跳转。注意，从库返回的操作不需要额外的开销。微软的 Windows 广泛地依赖动态链接库，如今在 UNIX 系统中程序执行的默认方式也是使用动态链接库。

2.12.6 启动一个 Java 程序

前面讨论了程序执行的传统模式，重点是以一个特定的指令集体系结构甚至这个体系结构的特定实现为目标的程序的快速执行。实际上，可以像 C 那样来执行 Java 程序。然而，Java 是为了不同的目标而发明的，其中之一就是能够安全地运行在每台计算机上，尽管这可能延长执行时间。

图 2-23 展示了典型的 Java 翻译和运行步骤。Java 程序会首先被编译成易于解释的指令序列——Java 字节码（Java bytecode）指令集（见 2.15 节），而不是编译成目标计算机可识别的汇编语言。这个指令集被设计得非常接近 Java 语言，这样，编译步骤相对简单，事实上它没有做任何优化。就像 C 语言编译器那样，Java 编译器会检查数据类型并且为每种类型提供正确的操作。Java 程序将转化成这些字节码的二进制形式。

- Java 字节码：为了解释 Java 程序而设计的指令集中的指令。

一个叫作 Java 虚拟机（Java Virtual Machine, JVM）的软件解释器能够执行 Java 字节码文件。解释器是一个用来模拟指令集体系结构的程序。例如，本书所使用的 MIPS 模拟器就是一个解释器。由于翻译非常简单，所以地址可以由编译器填写或在运行时被 JVM 发现，不需要再单独进行汇编。

- Java 虚拟机：解释 Java 字节码的程序。

解释的优势是可移植性。软件实现的 Java 虚拟机的可用性意味着在 Java 公布以后，大部分人都可以立即编写和运行 Java 程序。今天 Java 虚拟机可以用在从手机到网络浏览器等数亿的设备中。

解释的不足是性能较差。20 世纪 80 年代和 90 年代解释在执行性能上的飞速提高使它可用于很多重要的应用程序，但是与传统的编译好的 C 程序相比，10 倍的性能差距使 Java 对一些应用程序毫无吸引力。

为了既保持可移植性又提高执行速度，开发 Java 的下一阶段目标是实现程序执行的同时可以进行翻译的编译器。这个即时编译器（Just In Time compiler）通过记录运行的程序来找到称为“热点”的方法，然后将它们直接编译成 Java 虚拟机运行的宿主机的指令序列，编译过的部分保存起来以便下次程序运行时调用，这样，以后每次运行会更快。解释和编译的平衡随着时间的推移逐步形成，届时，经常运行的 Java 程序的解释开销变得非常小。

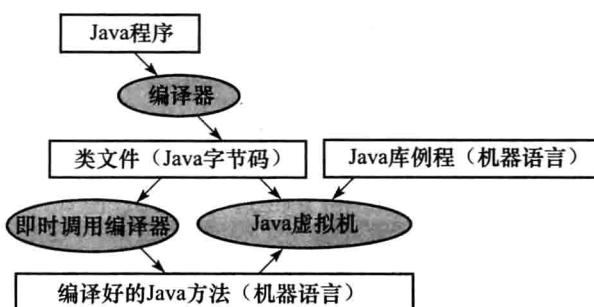


图 2-23 Java 的翻译层次。一个 Java 程序首先被编译成一个二进制版本的 Java 字节码形式，其中由编译器定义所有的地址。此时，Java 程序已可在解释器上运行，称为 Java 虚拟机（JVM）。在程序运行的时候，JVM 链接 Java 库中一些需要调用的方法。为了得到更好的性能，JVM 能够调用即时（just in time, JIT）编译器，在运行它的机器上能够选择性地把一些方法编译成宿主机上的本地机器语言

- 即时编译器：一类通用编译器的名称，编译器能够在运行时将解释的代码段翻译成宿主计算机上的机器语言。

随着计算机的速度越来越快，编译器能做的事情也越来越多。而随着研究者不断地发明更好的技术来编译 Java 程序，Java 与 C 或 C++ 在性能上的差距越来越小。2.15 节将进一步介绍 Java 程序、Java 字节码、JVM 和 JIT 编译器。

01 小测验

对 Java 设计者来说，你认为与翻译器相比，解释器在哪些方面的优点是最重要的？

1. 解释器便于编写。
2. 更准确的错误消息。
3. 更少的目标代码。
4. 机器独立性。

2.13 以一个 C 排序程序作为完整的例子

以片断的方式展示汇编代码的危险之处在于，你无法知道整个汇编语言程序的全貌。本节，我们给出了两个 C 过程对应的 MIPS 代码：一个用于交换（swap）数组的元素，另一个用于对数组元素排序（sort）。

2.13.1 swap 过程

我们从图 2-24 中的过程 swap 开始。这个过程简单地交换内存中两个位置的内容。我们按照以下常见的步骤把它从 C 程序手动翻译为汇编程序：

- 1) 为程序变量分配寄存器。
- 2) 为过程体生成汇编代码。
- 3) 保存过程调用间的寄存器。

本小节将按照这三个步骤描述 swap 程序，在最后把它们总结在一起。

1. 为 swap 分配寄存器

如 2.8 节所述，在 MIPS 中，实现参数传递通常使用寄存器 \$a0、\$a1、\$a2、\$a3。由于 swap 只需要两个参数，v 和 k，它们将被分配在寄存器 \$a0 和 \$a1 中。由于 swap 是一个叶过程（见 2.8.2 节），所以我们为唯一的剩余变量 temp 分配寄存器 \$t0。这些寄存器的分配与图 2-24 中的 swap 过程的第一部分变量的声明相对应。

2. 为 swap 过程体生成代码

swap 剩余部分的 C 代码如下所示：

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

回忆一下 MIPS 是按字节在内存中寻址的，字由 4 字节组成。因此我们需要把 k 乘以 4，再与地址相加。忘记连续的字之间的地址相差 4 而不是 1，是汇编语言程序设计中常见的错误。因此获得 v [k] 地址的第一步就是通过左移 2 位来使 k 乘以 4：

```
void swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

图 2-24 一个交换内存中两个不同位置所存的数值的 C 过程。本小节要在排序的例子中使用这个过程

```
sll    $t1, $a1,2      # reg $t1 = k * 4
add    $t1, $a0,$t1    # reg $t1 = v + (k * 4)
                  # reg $t1 has the address of v[k]
```

接下来使用 \$t1 来取 v [k] 的值，在使 \$t1 加 4 得到 v [k+1] 的地址：

```
lw     $t0, 0($t1)    # reg $t0 (temp) = v[k]
lw     $t2, 4($t1)    # reg $t2 = v[k + 1]
                  # refers to next element of v
```

最后将 \$t0 和 \$t2 存储到需要交换数据的地址中：

```
sw     $t2, 0($t1)    # v[k] = reg $t2
sw     $t0, 4($t1)    # v[k+1] = reg $t0 (temp)
```

至此，我们已经为该过程分配了寄存器并翻译好了程序体的代码。保存在 swap 中使用的保存寄存器的代码还没有完成。但是，由于这是一个叶过程，并没有使用保存寄存器，所以没有需要保留的东西。

3. 完整的 swap 程序

现在我们已经得到完整的例程了，包括程序标号和返回的跳转。为了方便读者的理解，在图 2-25 中，我们标明了过程中每个代码块的目的。

过程体
swap: sll \$t1, \$a1,2 # reg \$t1 = k * 4 add \$t1, \$a0,\$t1 # reg \$t1 = v + (k * 4) # reg \$t1 has the address of v[k]
lw \$t0, 0(\$t1) # reg \$t0 (temp) = v[k] lw \$t2, 4(\$t1) # reg \$t2 = v[k + 1] # refers to next element of v
sw \$t2, 0(\$t1) # v[k] = reg \$t2 sw \$t0, 4(\$t1) # v[k+1] = reg \$t0 (temp)
过程返回
jr \$ra # return to calling routine

图 2-25 图 2-24 中 swap 过程的 MIPS 汇编代码

2. 13.2 sort 过程

为保证你能够认识到汇编语言编程的严格性，我们提供了第二个更长的例子。在这个例子中，我们将编写一个调用 swap 过程的例程。这个例程对数组中的整数进行排序，使用的是冒泡或交换排序算法，这种排序算法虽然不是最快的，但却是最简单的。图 2-26 给出了该程序的 C 代码。我们还是使用几个步骤来演示翻译的过程，最后再把它们总结到一起。

```
void sort (int v[], int n)
{
    int i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
            swap(v, j);
        }
    }
}
```

图 2-26 一个对数组 v 中元素进行排序的 C 程序

1. sort 的寄存器分配

为过程 sort 的两个参数 v 和 n 分配参数寄存器 \$a0 和 \$a1，为变量 i 和 j 分别分配寄存器 \$s0 和 \$s1。

2. 为 sort 过程体生成代码

过程体包含两个嵌套的 for 循环和一个有参数的 swap 调用。我们将从外到内来展开代码。第一步来翻译最外面的 for 循环。

```
for (i = 0; i <n; i += 1) {
```

回忆 C 语言中 for 的声明有三个参数：初始值、循环判断条件和迭代增量。for 语句的第一部分是将 i 初始化为 0，这需要一条指令，

```
move $s0, $zero # i = 0
```

(请记住 move 是为了方便汇编程序员而由汇编器提供的伪指令，见 2.12.2 节。) for 语句的最后一部分，需要一条语句来增加 i：

```
addi $s0, $s0, 1 # i += 1
```

循环要在条件 $i < n$ 非真的时候退出，换句话说，当 $i \geq n$ 时循环退出。如果 $\$s0 < \$a1$ ，那么小于则置位指令将 $\$t0$ 置 1，否则置 0。因为我们要测试 $\$s0 \geq \$a1$ ，所以当寄存器 $\$t0$ 为 0 时，执行分支指令。这需要两条指令：

```
for1st:slt $t0, $s0, $a1 # reg $t0 = 0 if $s0 >= $a1 (i >= n)
      beq $t0, $zero, exit1 # go to exit1 if $s0 >= $a1 (i >= n)
```

循环的底部仅仅需要跳回循环判断的地方：

```
j for1st # jump to test of outer loop
exit1:
```

第一个 for 循环的框架代码为

```
move $s0, $zero # i = 0
for1st:slt $t0, $s0, $a1 # reg $t0 = 0 if $s0 >= $a1 (i >= n)
      beq $t0, $zero, exit1 # go to exit1 if $s0 >= $a1 (i >= n)
      .
      .
      (body of first for loop)
      .
      addi $s0, $s0, 1 # i += 1
j for1st # jump to test of outer loop
exit1:
```

(后面的练习将会进一步探索为类似的循环编写更快的代码。)

第二个 for 循环的 C 语句如下：

```
for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {
```

这个循环的初始化部分仍然是一条指令：

```
addi $s1, $s0, -1 # j = i - 1
```

循环末尾 j 的自减（减 1）也是一条指令：

```
addi $s1, $s1, -1 # j -= 1
```

循环判断由两个部分组成。任何一个条件为假就退出循环，所以第一个条件如果为假 ($j < 0$) 就要退出循环：

```
for2tst: slti $t0, $s1, 0 # reg $t0 = 1 if $s1 < 0 (j < 0)
      bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)
```

这将跳过第二个条件测试，如果没有跳过，则 $j \geq 0$ 。

第二个测试条件当 $v[j] > v[j + 1]$ 非真的时候退出，或 $v[j] \leq v[j + 1]$ 时退出。为得到地址，我们首先将 j 乘以 4（我们需要字节地址），然后将它与 v 的基址相加：

```
sll $t1, $s1, 2 # reg $t1 = j * 4
add $t2, $a0, $t1 # reg $t2 = v + (j * 4)
```

现在取 $v[j]$ ：

```
lw $t3, 0($t2) # reg $t3 = v[j]
```

因为我们知道第二个元素恰好是下一个字，所以我们将寄存器 $\$t2$ 值加 4，得到 $v[j + 1]$

135

136

的地址：

```
lw    $t4, 4($t2)  # reg $t4 = v[j + 1]
```

测试 $v[j] \leq v[j+1]$ 与测试 $v[j+1] \geq v[j]$ 相同，所以测试退出的两条指令如下：

```
slt  $t0, $t4, $t3  # reg $t0 = 0 if $t4 >= $t3  
beq $t0, $zero, exit2 # go to exit2 if $t4 >= $t3
```

循环末尾跳回到内层循环测试处：

```
j   for2tst  # jump to test of inner loop
```

将这些片段组合到一起，可得第二个 for 循环的框架如下：

```
addi $s1, $s0, -1      # j = i - 1  
for2tst: slti $t0, $s1, 0      # reg $t0 = 1 if $s1 < 0 (j < 0)  
        bne $t0, $zero, exit2 # go to exit2 if $s1 < 0 (j < 0)  
        sll $t1, $s1, 2      # reg $t1 = j * 4  
        add $t2, $a0, $t1      # reg $t2 = v + (j * 4)  
        lw   $t3, 0($t2)      # reg $t3 = v[j]  
        lw   $t4, 4($t2)      # reg $t4 = v[j + 1]  
        slt $t0, $t4, $t3      # reg $t0 = 0 if $t4 >= $t3  
        beq $t0, $zero, exit2 # go to exit2 if $t4 >= $t3  
        . . .  
        (body of second for loop)  
        . . .  
        addi $s1, $s1, -1      # j -= 1  
        j   for2tst          # jump to test of inner loop
```

exit2:

3. sort 中的过程调用

下一步翻译第二个 for 循环的循环体：

```
swap(v,j);
```

调用 swap 很容易：

```
jal   swap
```

4. sort 中的参数传递

当我们想传递参数时问题出现了，因为 sort 过程需要使用寄存器 \$a0 和 \$a1 中的值，而 swap 过程需要将它的参数放入这些寄存器。一种解决办法是在过程执行的早期将 sort 的参数复制到其他的寄存器中，使 swap 过程可以使用寄存器 \$a0 和寄存器 \$a1。（这个复制的过程比在栈中保存后再取回要快得多。）在过程中我们首先将寄存器 \$a0 和 \$a1 的值复制到寄存器 \$s2 和 \$s3。

```
move $s2, $a0      # copy parameter $a0 into $s2  
move $s3, $a1      # copy parameter $a1 into $s3
```

然后用下面两条指令将参数传递给 swap

```
move $a0, $s2      # first swap parameter is v  
move $a1, $s1      # second swap parameter is j
```

5. 在 sort 中保留寄存器

仅剩保存和恢复寄存器值的代码了。因为 sort 是一个过程并且它要递归使用，所以很明显需要用寄存器 \$ra 保存返回地址。sort 过程还使用了 \$s0、\$s1、\$s2 和 \$s3 保存寄存器，它们的值也必须被保存。所以 sort 过程头如下：

```
addi $sp,$sp,-20  # make room on stack for 5 registers  
sw   $ra,16($sp)  # save $ra on stack  
sw   $s3,12($sp)  # save $s3 on stack  
sw   $s2, 8($sp)   # save $s2 on stack  
sw   $s1, 4($sp)   # save $s1 on stack  
sw   $s0, 0($sp)   # save $s0 on stack
```

过程末尾只需反向执行这些指令，然后为了返回加上 jr 指令。

6. 完整的 sort 过程

现将所有片段合起来放入图 2-27，注意 for 循环中对寄存器 \$a0 和 \$a1 的引用已经被替换为对寄存器 \$s2 和 \$s3 的引用。为了方便阅读，我们再一次将过程中每一块的用途标了出来。本例中，9 行 C 语言编写的 sort 过程被翻译成 35 行的 MIPS 汇编语言代码。

保存寄存器值	
<pre>sort: addi \$sp, \$sp, -20 # make room on stack for 5 registers sw \$ra,16(\$sp) # save \$ra on stack sw \$s3,12(\$sp) # save \$s3 on stack sw \$s2,8(\$sp) # save \$s2 on stack sw \$s1,4(\$sp) # save \$s1 on stack sw \$s0,0(\$sp) # save \$s0 on stack</pre>	
过程体	
移动参数	<pre>move \$s2, \$a0 # copy parameter \$a0 into \$s2 (save \$a0) move \$s3, \$a1 # copy parameter \$a1 into \$s3 (save \$a1)</pre>
循环外部	<pre>move \$s0, \$zero # i = 0 forltst: slt \$t0, \$s0, \$s3 # reg \$t0 = 0 if \$s0 < \$s3 (i < n) beq \$t0, \$zero, exit1 # go to exit1 if \$s0 < \$s3 (i < n)</pre>
循环内部	<pre>addi \$s1, \$s0, -1 # j = i - 1 for2tst: slti \$t0, \$s1, 0 # reg \$t0 = 1 if \$s1 < 0 (j < 0) bne \$t0, \$zero, exit2 # go to exit2 if \$s1 < 0 (j < 0) sll \$t1, \$s1, 2 # reg \$t1 = j * 4 add \$t2, \$s2, \$t1 # reg \$t2 = v + (j * 4) lw \$t3, 0(\$t2) # reg \$t3 = v[j] lw \$t4, 4(\$t2) # reg \$t4 = v[j + 1] slt \$t0, \$t4, \$t3 # reg \$t0 = 0 if \$t4 < \$t3 beq \$t0, \$zero, exit2 # go to exit2 if \$t4 < \$t3</pre>
传递参数和调用	<pre>move \$a0, \$s2 # 1st parameter of swap is v (old \$a0) move \$a1, \$s1 # 2nd parameter of swap is j jal swap # swap code shown in Figure 2.25</pre>
循环内部	<pre>addi \$s1, \$s1, -1 # j -= 1 j for2tst # jump to test of inner loop</pre>
循环外部	<pre>exit2: addi \$s0, \$s0, 1 # i += 1 j forltst # jump to test of outer loop</pre>
恢复寄存器的值	
<pre>exit1: lw \$s0,0(\$sp) # restore \$s0 from stack lw \$s1,4(\$sp) # restore \$s1 from stack lw \$s2,8(\$sp) # restore \$s2 from stack lw \$s3,12(\$sp) # restore \$s3 from stack lw \$ra,16(\$sp) # restore \$ra from stack addi \$sp, \$sp, 20 # restore stack pointer</pre>	
过程返回	
<pre>jr \$ra # return to calling routine</pre>	

图 2-27 图 2-26 中 sort 过程的 MIPS 汇编版本

01 精解 这个例子可以使用的一种优化方法是内联过程（procedure inlining）。在代码中调用 swap 过程的地方，编译器将 swap 的过程体的代码复制过来，而不是通过传递参数并通过 jal 指令来调用这段代码。本例中使用内联可以省掉 4 条指令。使用内联优化的缺点是如果内联过程需要在多个地方调用，编译后产生的代码将会变多。如果这种代码扩展导致 cache 的缺失率上升，将导致性能的下降（见第 5 章）。

01 理解程序性能 图 2-28 展示了编译器优化对排序程序的性能、编译时间、时钟周期、指令数和 CPI 的影响。注意没有优化的代码具有最好的 CPI，使用 O1 优化的代码具有最少的指令数，但是 O3 优化的执行速度最快，这告诉我们执行时间是准确衡量程序性能的唯一指标。

图 2-29 比较了编程语言、编译执行或解释执行和算法对排序程序性能的影响。第四列表明在执行冒泡排序时没有优化的 C 程序比解释型的 Java 程序快 8.3 倍。使用即时编译器可以使 Java 比没有优化的 C 程序快 2.1 倍，比最佳优化的 C 代码慢不到 1.13 倍。（2.15 节将给出关于解释执行和编译执行 Java 的更多细节以及冒泡排序的 Java 和 MIPS 代码。）在第五列中，快速排序的性能比就沒那么接近了，这大概是因为在这样短的执行时间内分摊运行时编译的时间是非常困难的。最后一列展示了更好的算法带来的影响，当对 100 000 个元素进行排序时，性能达到了 3 个数量级的提升。即第五列中解释执行的 Java 与第四列中最优化的 C 代码相比，快速排序法要比冒泡法快 50 倍 (0.05×2468 或者用 $123/2.41$)。

gcc 优化选项	相对性能	时钟周期（百万）	指令数（百万）	CPI
无	1.00	158 615	114 938	1.38
O1（中等）	2.37	66 990	37 470	1.79
O2（完全）	2.38	66 521	39 993	1.66
O3（过程集成）	2.41	65 747	44 993	1.46

图 2-28 冒泡排序中编译器优化对性能、指令数、CPI 的影响比较。程序对含有 100 000 个字的被初始化为随机数的数组进行排序。程序运行在 3.06GHz 的奔腾 4 处理器上，前端系统总线是 533MHz，具有 2GB 的 PC2100 DDR SDRAM。操作系统使用 Linux 2.4.20

编程语言	执行方式	优化选项	冒泡排序 相对性能	快速排序 相对性能	快速排序相对冒泡 排序加速比
C	编译器	无	1.00	1.00	2 468
	编译器	O1	2.37	1.50	1 562
	编译器	O2	2.38	1.50	1 555
	编译器	O3	2.41	1.91	1 955
Java	解释器	—	0.12	0.05	1 050
	即时编译器	—	2.13	0.29	338

图 2-29 两个排序算法的性能比较。算法分别用 C 和 Java 实现，Java 分别使用解释执行和优化编译来与未优化的 C 版本比较。最后一列是快速排序比冒泡排序在每种语言和执行方式下速度提高多少。这些程序运行的系统与图 2-28 相同。JVM 是 Sun 的 1.3.1 版本，JIT 是 Sun Hotspot 的 1.3.1 版本

01 精解 MIPS 的编译器总是在栈上为参数保留空间以便它们得以保存，所以实际上 \$sp 总是减 16 来给 4 个参数寄存器（16 字节）分配空间。这样做的原因是 C 提供一个 vararg 选项，该选项允许选择一个指针，例如过程的第三个参数。当编译器遇到这种少见的 vararg 时，它就将 4 个参数寄存器的值都复制到栈上已经保留的位置中。

2.14 数组与指针

理解指针对任何一个C程序新手来说都是具有挑战的。通过对比使用数组和数组标记的汇编代码和使用指针的汇编代码，可以从本质上理解指针。本节将展示C和MIPS汇编版本的两个清除内存中连续字的过程：一个使用数组标记；另一个使用指针。图2-30给出了这两个C过程。

```
clear1(int array[], int size)
{
    int i;
    for (i = 0; i < size; i += 1)
        array[i] = 0;
}
clear2(int *array, int size)
{
    int *p;
    for (p = &array[0]; p < &array[size]; p = p + 1)
        *p = 0;
}
```

图2-30 两个将数组清零的C过程。`clear1` 使用下标，而`clear2` 使用指针。对不熟悉C的人，第二个过程需要做一些解释。变量的地址使用`&`表示，指针所指向的对象用`*`表示。声明部分说明`array`和`p`都是指向整数的指针。`clear2`的`for`循环中第一个部分将`array`的第一个元素的地址赋值给指针`p`。`for`循环的第二部分判断这个指针是否指向了`array`的最后一个元素之外。`for`循环的最后一部分，对这个指针每次递增（增1），意味着将指针移到它声明的空间中的下一个对象。由于`p`是一个指向整数的指针，编译器将会产生MIPS指令，让`p`按照4递增，4是MIPS中整数的字节数目。循环体中将0赋值给`p`所指向的对象。

本节的目的是展示指针是如何映射到MIPS指令的，而不是赞同这种过时的编程风格。我们在本节的末尾将看到现代编译器的优化对这两个过程带来的影响。

2.14.1 用数组实现clear

我们从数组版本的`clear1`开始，主要关注循环体，而忽略过程链接相关的代码。假设两个参数`array`和`size`分别在寄存器`$a0`和`$a1`中，`i`保存在`$t0`中。

for循环的第一部分，初始化变量`i`:

```
move $t0,$zero      # i = 0 (register $t0 = 0)
```

为了将`array [i]`清0，我们首先需要得到它的地址。首先把`i`乘以4得到字节地址：

```
loop1: sll $t1,$t0,2      # $t1 = i * 4
```

因为数组的起始地址在寄存器中，所以我们必须将它与下标相加以得到`array [i]`的地址，使用下面的加法指令：

```
add $t2,$a0,$t1      # $t2 = address of array[i]
```

然后，我们就将0保存在这个地址：

```
sw $zero, 0($t2)      # array[i] = 0
```

这条指令是循环体最后一条指令，下一步是增加`i`值（加1）：

```
addi $t0,$t0,1      # i = i + 1
```

循环测试条件检测`i`是否小于`size`：

```
slt $t3,$t0,$a1      # $t3 = (i < size)
bne $t3,$zero,loop1  # if (i < size) go to loop1
```

现在，我们已经得到过程所有的片断。下面则是使用数组下标对数组清零的MIPS汇编编码：

```

move $t0,$zero      # i = 0
loop1: sll $t1,$t0,2    # $t1 = i * 4
add $t2,$a0,$t1      # $t2 = address of array[i]
sw $zero, 0($t2)      # array[i] = 0
addi $t0,$t0,1        # i = i + 1
slt $t3,$t0,$a1        # $t3 = (i < size)
bne $t3,$zero,loop1   # if (i < size) go to loop1

```

(只要 size 大于 0, 这些代码就能正确工作; ANSI C 需要在循环前测试 size 值, 但是我们跳过了这个。)

142

2.14.2 用指针实现 clear

第二个过程是使用指针的, 该过程将两个参数 array 和 size 分配到寄存器 \$a0 和 \$a1, 将 p 分配到寄存器 \$t0。在第二个过程开始时需要将数组的首地址赋值给指针 p:

```
move $t0,$a0          # p = address of array[0]
```

接下来的代码将是 for 循环体, 它仅仅是简单地将 0 存到地址 p:

```
loop2: sw $zero,0($t0)  # Memory[p] = 0
```

这条指令实现了循环体, 所以下一条指令将是迭代子自增, 即改变 p 使其指向下一个字:

```
addi $t0,$t0,4        # p = p + 4
```

在 C 中将指针加 1 意味着将指针指向序列中下一个对象。因为 p 是一个指向整数的指针, 整数占用 4 字节, 编译器将对 p 加 4。

接着就是循环测试。首先计算 array 最后一个元素的地址。先将 size 乘以 4 得到字节地址。

```
sll $t1,$a1,2          # $t1 = size * 4
```

然后, 将乘积与数组的首地址相加以获得数组后面第一个字的地址:

```
add $t2,$a0,$t1        # $t2 = address of array[size]
```

循环测试仅仅是简单地判断 p 是否比 array 最后一个元素的地址小:

```
slt $t3,$t0,$t2        # $t3 = (p < &array[size])
```

```
bne $t3,$zero,loop2    # if (p < &array[size]) go to loop2
```

所有的代码片段都已经完成, 现在我们可以看到指针版本的数组清零了:

```

move $t0,$a0          # p = address of array[0]
loop2: sw $zero,0($t0)  # Memory[p] = 0
addi $t0,$t0,4        # p = p + 4
sll $t1,$a1,2          # $t1 = size * 4
add $t2,$a0,$t1        # $t2 = address of array[size]
slt $t3,$t0,$t2        # $t3 = (p < &array[size])
bne $t3,$zero,loop2    # if (p < &array[size]) go to loop2

```

与第一个例子一样, 这段代码也假定 size 大于 0。

注意, 尽管数组的末地址一直保持不变, 但是这个程序循环的每次迭代都要计算它。一种快速的执行方式是将数组末地址的计算放到循环体外面:

```

move $t0,$a0          # p = address of array[0]
sll $t1,$a1,2          # $t1 = size * 4
add $t2,$a0,$t1        # $t2 = address of array[size]
loop2: sw $zero,0($t0)  # Memory[p] = 0
addi $t0,$t0,4        # p = p + 4
slt $t3,$t0,$t2        # $t3 = (p < &array[size])
bne $t3,$zero,loop2    # if (p < &array[size]) go to loop2

```

143

2.14.3 比较两个版本的 clear

将两段代码放在一起进行比较可以说明数组下标和指针的不同 (指针版本带来的变化被高

亮显示)：

```

move $t0,$zero      # i = 0
loop1:s11 $t1,$t0,2    # $t1 = i * 4
add  $t2,$a0,$t1    # $t2 = &array[i]
sw   $zero, 0($t2)  # array[i] = 0
addi $t0,$t0,1      # i = i + 1
slt  $t3,$t0,$a1    # $t3 = (i < size)
bne $t3,$zero,loop1# if () go to loop1
move $t0,$a0      # p = &array[0]
sll  $t1,$a1,2    # $t1 = size * 4
add  $t2,$a0,$t1    # $t2 = &array[size]
loop2:sw $zero,0($t2) # Memory[p] = 0
addi $t0,$t0,4      # p = p + 4
slt  $t3,$t0,$t2    # $t3=(i<&array[size])
bne $t3,$zero,loop2# if () go to loop2

```

左边的版本必须在循环中有“乘”和加操作，因为 *i* 值增加了，每个地址都将从新下标开始被重新计算。右边存储器指针版本的代码直接增加指针 *p*。指针版本通过把一些操作拿到循环外部，将每次迭代执行的指令从 6 条减少到 4 条。这种手动的优化与编译器的强度减少(用移位代替乘)和变量消除(消除循环中的数组地址计算)是一致的。2.15 节叙述了这两种优化和其他一些优化。

01 精解 正如前面提到的，C 编译器需要增加测试来保证 *size* 一定大于 0。一个方法是在循环的第一条指令之前加入一条跳转到 *slt* 的跳转指令。

01 理解程序性能 以往经常教育人们要在 C 中使用指针来获得数组所无法获得的更高的效率。然而，“使用指针，甚至会使你自己都无法理解代码的含义。”现代的优化编译器可以为数组版本产生同样好的代码。现在大部分程序员更喜欢让编译器去做更繁重的工作。

2.15 高级内容：编译 C 语言和解释 Java 语言

本节将简要概述 C 编译器如何工作和 Java 是如何执行的。因为编译器将对计算机的性能产生重要影响，所以理解编译器技术是理解性能的关键。要知道“编译器的构建”课程的学习一般需要 1 个或 2 个学期，所以我们这里将仅仅介绍一些基本内容。

本节的第二部分是为对面向对象语言(objected oriented language)(例如 Java)在 MIPS 体系结构上执行感兴趣的读者准备的。本节将展示被用于解释执行的 Java 字节码和前面章节中用 C 编写的程序段的 Java 版本的 MIPS 代码，包括冒泡排序。本节将包括 Java 虚拟机和即时编译器。

本节的剩余内容在配套网站上。

② 面向对象语言：一种针对对象而不是动作的编程语言，或者针对数据而不是逻辑的编程语言。

2.16 实例：ARMv7 (32 位) 指令集

在嵌入式设备领域中最流行的指令集体系结构是 ARM，2011 年有超过 90 亿部各种各样的设备使用 ARM 处理器，并且以每年 20 亿的数量增长。ARM 最初代表 Acorn RISC Machine，稍后被改为 Advanced RISC Machine。ARM 与 MIPS 处理器在同年发布并遵循相同的设计哲学。图 2-31 列出了 ARM 与 MIPS 的相似性。它们二者的主要区别是 MIPS 有更多的寄存器，而 ARM 有更多的寻址模式。

图 2-32 展示了 MIPS 与 ARM 在算术逻辑和数据传输指令方面具有相似的核心指令集。

	ARM	MIPS
发布时间	1985	1985
指令大小（位）	32	32
寻址空间（大小，模式）	32 位，平坦	32 位，平坦
数据对齐	对齐	对齐
数据寻址模式	9	3
整数寄存器（个数，模式，大小）	15 通用寄存器 × 32 位	31 通用寄存器 × 32 位
I/O	存储器映射	存储器映射

图 2-31 ARM 和 MIPS 指令集的相同点

指令名	ARM	MIPS
寄存器 - 寄存器	加法	add
	加法（溢出捕获）	adds; swivs
	减法	sub
	减法（溢出捕获）	subs; swivs
	乘法	mul
	除法	—
	与	and
	或	orr
	异或	eor
	取寄存器高位	—
	逻辑左移	lsl ¹
	逻辑右移	lsr ¹
	算术右移	asr ¹
	比较	cmp, cmn, tst, teq
数据传输	取有符号字节	ldr sb
	取无符号字节	ldr b
	取有符号半字	ldr sh
	取无符号半字	ldr h
	取字	ldr
	存字节	str b
	存半字	str h
	存字	str
	读、写特殊寄存器	mrs, msr
	原子交换	swp, swpb

图 2-32 ARM 的寄存器 - 寄存器指令和数据传输指令与 MIPS 核心指令是等价的。横线表示体系结构不支持该操作或不能用一些指令来实现该操作。如果有几条可供选择的指令都与 MIPS 核心指令等价，那么用逗号分隔这些指令。ARM 中每条数据操作指令都有移位的部分，所以移位指令用了上标 1，它们基本是 move 指令的变种，例如 lsr¹。注意 ARM 中没有除法指令

2.16.1 寻址模式

图 2-33 展示了 ARM 支持的数据寻址模式。不同于 MIPS，ARM 不需要使用专门的寄存器来保存 0 这个数值。尽管 MIPS 仅有 3 种简单的数据寻址模式（见图 2-18），ARM 却有 9 种寻址模式之多，包括十分复杂的计算的寻址模式。例如，ARM 的一种寻址模式可以把一个寄存器中的数移动任意位，将移位后得到的数与另外一个寄存器中的值相加产生地址，然后将产生

的新地址存入一个寄存器中。

寻址模式	ARM	MIPS
寄存器操作数	×	×
立即数操作数	×	×
寄存器 + 偏移（转移或基地址）	×	×
寄存器 + 寄存器（下标）	×	—
寄存器 + 寄存器倍乘（倍乘）	×	—
寄存器 + 偏移和更新寄存器	×	—
寄存器 + 寄存器和更新寄存器	×	—
自增，自减	×	—
相对 PC 的数据	×	—

图 2-33 数据寻址模式的总结。ARM 具有分离的寄存器间接寻址和寄存器 + 偏移寻址模式，而不是仅仅在后一种模式的偏移地址上填 0。为了增加寻址范围，如果是对半字或字进行操作，ARM 对偏移左移 1 位或 2 位

2.16.2 比较和条件分支

MIPS 使用寄存器中的值来决定条件分支是否执行。而 ARM 使用传统的存储在程序状态字中的 4 位条件码来决定条件分支是否执行。这 4 个条件码是：负值（negative）、零（zero）、进位（carry）和溢出（overflow）。这些条件码可以被任何算术或逻辑指令设置，不同于早期的体系结构，这些设置功能是每条指令的可选功能。明确的选项会使流水化的实现变得更加容易。ARM 使用条件分支来测试条件码以判断所有有符号和无符号的关系。

CMP 指令用一个操作数减去另一个操作数，用它们的差设置条件码。CMN 指令将一个操作数与另一个操作数相加，用它们的和来设置条件码。TST 指令将两个操作数进行逻辑与，然后设置除溢出位外其他的条件码。TEQ 指令是用异或结果来设置条件码的前三位。

ARM 具有这样一个不寻常的特点，每条指令都有一个可选的执行条件，这个条件决定于条件码。每条指令开始的 4 位字段决定这条指令将执行空操作（nop）还是执行真实的指令操作，这种选择也取决于条件码。因此，条件分支也可以被认为是有条件的执行无条件分支指令。条件执行指令可以取代仅为了跳过一条指令的分支指令，不仅占用的代码空间更少，而且也会节省运行时间。

图 2-34 展示了 ARM 和 MIPS 的指令格式。它们之间的主要区别有两点：每条指令的 4 位条件执行字段不同；ARM 因为只用 MIPS 一半数量的寄存器，所以具有相对较小的寄存器字段。

2.16.3 ARM 的特色

图 2-35 列举了 ARM 处理器所特有的一些算术逻辑指令，这些指令在 MIPS 中是不存在的。由于没有专门的寄存器用来存储 0，所以 ARM 需要单独的操作码来完成一些在 MIPS 中可以简单使用 \$zero 来完成的操作。另外，ARM 支持多个字的算术操作。

ARM 解释 12 位立即数字段的方式非常新颖。首先将右侧低 8 位的有效位填 0 扩展到 32 位，然后将所得的数循环右移，移动的位数由高 4 位的值乘以 2 决定。这种解释方式的优点是可以在 32 位字的范围内表达所有 2 的幂次。为什么这种分割所表示的数字多于简单的 12 位字段是一个有趣的问题。

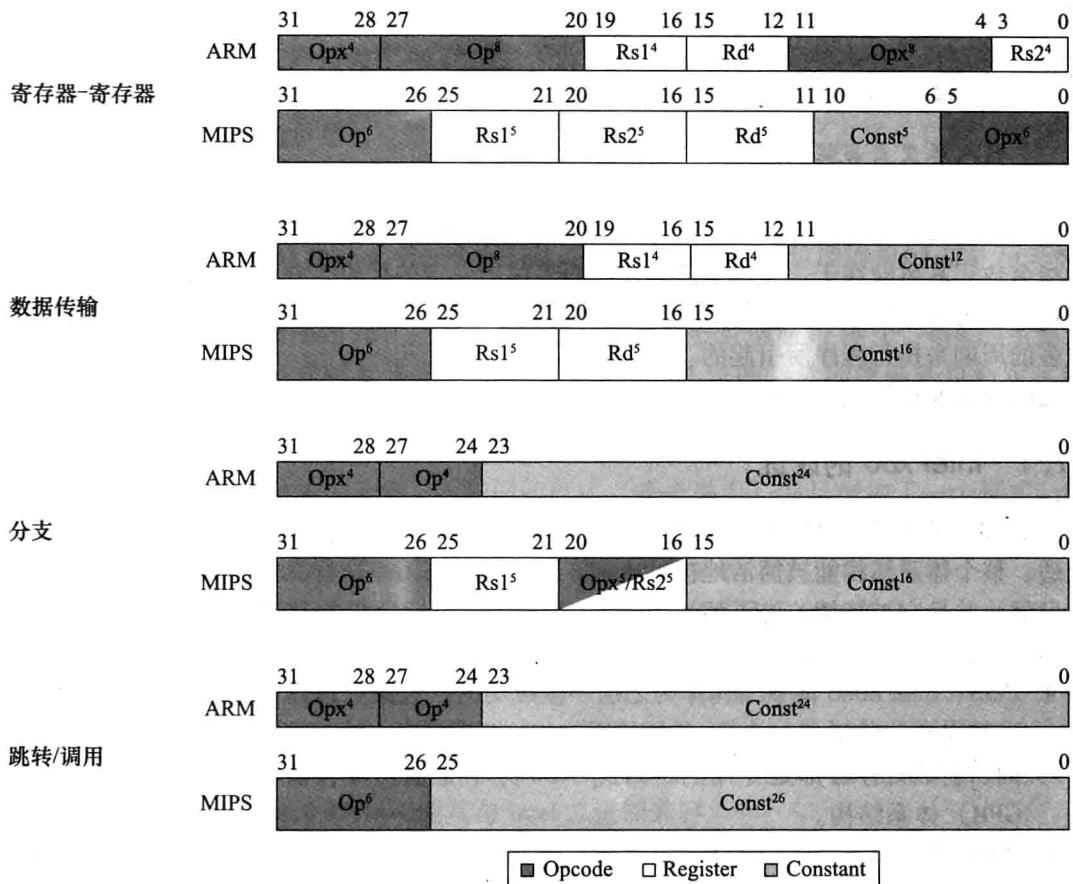


图 2-34 ARM 和 MIPS 的指令格式。区别在于体系结构中是有 16 个还是 32 个寄存器

名字	定义	ARM	MIPS
取立即数	Rd = Imm	mov	addi \$0,
非	Rd = ~ (Rs1)	mvn	nor \$0,
移动	Rd = Rs1	mov	or \$0,
循环右移	Rd = Rsi >> i Rd _{0...i-1} = Rs _{31-i...31}	ror	
寄存器与另一寄存器的非进行与操作	Rd = Rs1 & ~ (Rs2)	bic	
反向减	Rd = Rs2 - Rs1	rsb, rsc	
支持多个整数字的加	CarryOut, Rd = Rd + Rs1 + OldCarryOut	adcs	—
支持多个整数字的减	CarryOut, Rd = Rd - Rs1 + OldCarryOut	sbs	—

图 2-35 MIPS 中没有的 ARM 算术/逻辑指令

对操作数的移位并不仅限于立即数。所有算术和逻辑运算操作的第二个寄存器操作数都可以在执行操作之前进行移位。可选的移位方式是逻辑左移、逻辑右移、算术右移和循环右移。

ARM 还对寄存器组的操作提供了指令支持，这些指令叫作块加载和存储（block loads and stores）。在指令的 16 位掩码的控制下，16 个寄存器中的任意组合都可以被一条指令加载或存储到内存中。这些指令可以保存和恢复程序调用和返回时的寄存器。这些指令也可以被用于存

储器块的复制，现在这些存储器块的复制是对这些指令的主要应用。

2.17 实例：x86 指令集

情人眼里出西施。

——Margaret Wolfe Hungerford, 《Molly Bawn》, 1877

指令集的设计者有时提供比 ARM 和 MIPS 更强大的操作。这样做的目的是减少程序需要执行的指令数。其风险在于，在设备简单性方面需付出一定的代价，并且可能使程序执行时间变长，这是因为指令执行需要更长的时间。这可能是由于时钟周期变长或者是比更简单的序列需要更多的周期来执行程序所引起的。

通向复杂操作的道路困难重重。2.19 节将阐述复杂性的陷阱。

2.17.1 Intel x86 的改进

ARM 和 MIPS 都是由单独的小组在 1985 年推出的。这种体系结构的每部分配合在一起非常合适，整个体系结构能被简洁地描述出来。但是 x86 却不是这样，它是由一些相互独立的小组开发的，并且它被持续改进了超过 35 年，不断在原来指令集的基础上增加新的特性，这就像有些人往包装好的包里添加衣服。下面是 x86 发展的一些重要的里程碑。

149

- 1978：Intel 8086 体系结构作为之前一款成功的 8 位微处理器的汇编语言的可兼容的扩展被发布。8086 是一个 16 位的体系结构，所有内部的寄存器都是 16 位长。与 MIPS 不同，它的寄存器都是专用的，因此 8086 并不是通用寄存器（general-purpose register, GPR）体系结构。
- 1980：Intel 8087 浮点协处理器发布。这个体系结构在 8086 的基础上增加了 60 条浮点指令。它通过栈来代替寄存器（见 2.21 节和 3.7 节）。
- 1982：80286 在 8086 的基础上把地址空间扩展到 24 位，并设计了精妙的内存映射和保护模式（见第 5 章），还增加了一些指令去丰富整个指令集以及控制保护模式。
- 1985：80386 在 80286 体系结构的基础上将地址空间扩展到 32 位。除了 32 位的寄存器和 32 位的地址空间，80386 也增加了一些新的寻址模式和额外的操作。增加的指令使得 80386 几乎就是通用寄存器的处理器。80386 还增加了对页的支持并提供了段寻址（参见第 5 章）。与 80286 一样，80386 也提供能运行不经修改的 8086 程序的模式。
- 1989 ~ 1995：接下来在 1989 年发布了 80486，1992 年发布 Pentium 处理器，1995 年发布 Pentium Pro 处理器。这些处理器都是以获得更高的性能为目的的，仅有 4 个指令被增加到用户可见的指令集中，其中 3 个有助于多处理技术（参见第 6 章），另一个是条件传送指令。
- 1997：在 Pentium 和 Pentium Pro 销售后，Intel 公司宣称他们将用多媒体扩展 MMX（Multi Media Extension）来扩展 Pentium 和 Pentium Pro 的体系结构。这个新指令集包含 57 条指令，使用浮点栈来加速多媒体和通信应用程序。MMX 通过传统的单指令多数据（single instruction, multiple data, SIMD）的方式来一次处理多个短的数据元素（参见第 6 章）。Pentium II 没有引入任何新的指令。
- 1999：Intel 添加了另外 70 条指令，将 SSE（Streaming SIMD Extension）作为 Pentium III 的一部分。主要的变化是添加了 8 个独立的寄存器，把它们的长度增加到 128 位，并且增加了一个单精度浮点数据类型。因此，4 个 32 位的浮点操作就可以并行进行。为了改进内存性能，SSE 还包括 cache 的预取指令，以及可以绕过缓冲器直接写内存的流存储指令。

- 2001：Intel 公司增加了另外 144 条指令。这次命名为 SSE2。增加的新的数据类型是双精度算术，它允许并行操作 64 位浮点型数据对。这 144 条指令几乎都对应着一些已经存在的 MMX 和 SSE 指令，这些指令并行操作 64 位数据。这种变化不仅允许更多的多媒体操作，并且与单独的栈架构相比，编译器多了一个新的浮点操作目标。编译器可以使用 8 个 SSE 寄存器来充当浮点寄存器。这种改进大大增强了第一个包括 SSE2 指令集的微处理器 Pentium 4 的浮点性能。
- 2003：这次是 AMD 改进了 x86 体系结构，把地址空间从 32 位增加到 64 位。与 1985 年在 80386 上从 16 位到 32 位的转变类似，AMD64 把所有的寄存器都拓宽到 64 位，并且把寄存器的数目增加到 16，把 128 位的 SSE 寄存器数目增加到 16 个。ISA 的主要变化是新增加了一个模式叫长模式（long mode），用 64 位的地址和数据来重新定义所有 x86 指令的执行。为了寻址更多的寄存器，给指令增加了新前缀。根据计算方式，长模式还添加了 4~10 条新的指令并且去掉了 27 条旧指令。PC 相对数据寻址是另一个扩展。AMD64 仍然有一个和 x86 相同的模式（遗产模式）并且增加了一个模式，以限制用户程序使用 x86 模式，但是却允许操作系统使用 AMD64 模式（兼容模式）。这些模式使它成为比 HP/Intel IA-64 更好地从 32 位过渡到 64 位寻址的处理器。
- 2004：Intel 屈服并吸纳了 AMD64，重新标记为 Extended Memory64 Technology (EM64T)，主要的区别是 Intel 增加了 128 位的原子比较和交换指令，这个本应在 AMD64 上可能具有的指令。同时，Intel 发布了新一代媒体扩展。SSE3 添加了 13 条指令来支持复杂算术，包括在结构数组上进行的图形操作、视频编码、浮点转换以及线程同步（见 2.11 节）。AMD 会在以后的芯片中提供对 SSE3 的支持。而且它几乎肯定能够把原先没有的原子交换指令添加到 AMD64 使其与 Intel 二进制兼容。
- 2006：作为 SSE4 的一部分扩展，Intel 发布了 54 条新指令。这些扩展都是针对像如下影响性能的因素：绝对差求和、数组结构的点积计算、窄数据到较宽的数据的符号或零扩展，序列中非零的数目统计等。还增加了对虚拟机的支持（见第 5 章）。
- 2007：作为 SSE5 的一部分，AMD 发布了 170 条指令，包括为 46 条基本指令集中的指令增加了像 MIPS 的 3 操作数的版本。
- 2011：Intel 发布了高级向量扩展，同时将 SSE 寄存器从 128 位扩展到 256 位，因此重新定义了 250 条指令并新增了 128 条指令。

② 通用寄存器：可用于存储任何指令的地址或数据的寄存器。

这段历史说明了兼容性这个“金手铐”对 x86 的影响，体系结构的改变不允许对已有的软件产生任何的危害。

无论 x86 结构有多失败，该指令集一直对个人计算机的更新换代起着很大的推动作用，在后 PC 时代占据着很大的份额。表面上看起来，3.5 亿 x86 芯片的年产量相对于 ARMv7 芯片的 90 亿片要小很多，但是许多公司都想去控制这个市场。无论如何，这个多变的家族带来的是一个难以解释并且不讨人喜欢的体系结构。

请鼓起勇气来面对你将要看到的内容！不要带着需要编写 x86 程序的担心来阅读这一节，实际上，本节的目的是让你熟悉这一世界上最流行的台式机体系结构的优缺点。

本节我们主要关心的是 80386 的 32 位指令子集，而不是整个 16 位、32 位和 64 位指令集。我们从寄存器和寻址模式开始说明，接下来是整数操作，最后考虑指令编码。

2.17.2 x86 寄存器和数据寻址模式

80386 的寄存器展示了指令集的进化（如图 2-36 所示）。80386 把 16 位寄存器（除了段寄

[152] 存器) 扩展为 32 位。并用前缀 E 来标示 32 位版本。它们通常被称为通用寄存器。80386 只有 8 个通用寄存器, 这意味着 MIPS 程序使用 4 倍数量的寄存器, 而 ARMv7 可以使用 2 倍数量的寄存器。

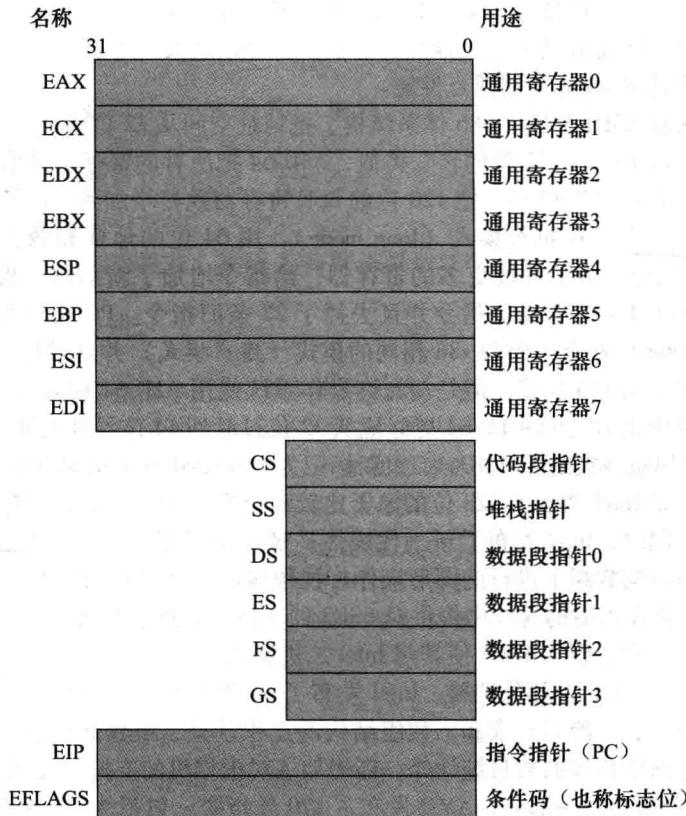


图 2-36 80386 寄存器组。从 80386 开始, 上面的 8 个寄存器扩展到 32 位并可以当做通用寄存器使用

图 2-37 展示了两个操作数的算术、逻辑和数据传输指令。它们有两个重要的不同之处。首先 x86 的算术和逻辑指令中的一个操作数必须既是源操作数又是目的操作数, 而 ARMv7 和 MIPS 的源操作数和目的操作数是不同的寄存器。这种限制给有限的寄存器带来更大的压力, 因此一个源寄存器必须被改变。第二个重要的不同之处在于一个操作数可以在存储器中。这样, 实质上任何指令都可能有一个操作数在存储器中。这与 ARMv7 和 MIPS 不同。

源/目标操作数类型	第二个源操作数
寄存器	寄存器
寄存器	立即数
寄存器	存储器
存储器	寄存器
存储器	立即数

图 2-37 算术、逻辑和数据传输指令的指令格式。x86 所允许的组合见上表。唯一的限制是没有存储器 - 存储器模式。立即数可以是 8 位、16 位或 32 位; 寄存器可以是图 2-36 中 14 个主要的寄存器 (不能是 EIP 或 EFLAGS) 的任意一个

后面将会详细阐述数据的存储器寻址模式, 在指令中提供两种位长的地址。这种所谓的偏移 (displacements) 既可能是 8 位也可能是 32 位。

尽管存储器操作数可以使用任何寻址模式, 但是每种模式使用哪些寄存器是有限制的。

图 2-38 展示了 x86 寻址模式和每种模式下哪个 GPR 是不允许使用的，并说明如何使用 MIPS 指令集来达到相同效果。

模式	描述	寄存器限制	等价的 MIPS
寄存器间接寻址	地址在寄存器中	不能为 ESP 或 EBP	<code>lw \$s0,0(\$s1)</code>
8 位或 32 位偏移寻址模式	地址是基址寄存器与偏移量之和	不能为 ESP	<code>lw \$s0,100(\$s1) #<=16 bit # displacement</code>
基址加比例下标寻址	地址是 基址 + (2 ^{比例} × 下标) 比例是 0, 1, 2 或 3	基址：任何 GPR 下标：不能为 ESP	<code>mul \$t0,\$s2, add \$t0,\$t0,\$s1 lw \$s0,0(\$t0)</code>
8 位或 32 位偏移量的基址 + 比例下标寻址	地址是 基址 + (2 ^{比例} × 下标) + 偏移量 比例是 0, 1, 2 或 3	基址：任何 GPR 下标：不能为 ESP	<code>mul \$t0,\$s2,4 add \$t0,\$t0,\$s1 lw \$s0,100(\$t0) #<=6-bit # displacement</code>

图 2-38 x86 有寄存器使用限制的 32 位寻址模式及等价的 MIPS 代码。ARM 和 MIPS 所没有的，基址加比例下标寻址模式，包含在 x86 中以避免将寄存器中的下标乘 4（使用比例因子 2）变成字节地址（见图 2-25 和图 2-27）。比例因子 1 用于 16 位数据，3 用于 64 位数据。比例因子 0 意味着这个地址不需要按比例增加。在第二种或第四种模式中如果偏移量比 16 位长，等价的 MIPS 需要额外的两条指令：lui 取偏移量的高 16 位，add 将高 16 位与寄存器 \$s1 相加。（Intel 的基址寻址模式还有另外的名字基址和下标，但是它们本质上是等同的，我们在这里将它们合并。）

2.17.3 x86 整数操作

8086 提供对 8 位（字节）和 16 位（字）数据类型的支持。80386 在 x86 结构中加入了 32 位的地址和数据（双字）。（AMD64 又增添了 64 位的地址和数据，叫作四字；本小节我们将关注 80386。）数据类型的不同也造成了寄存器操作和存储器访问的不同。

几乎所有操作都能在 8 位和一个更长的数据上进行。这个最长的数据大小取决于运行的模式，可能是 16 位也可能是 32 位。

显然，有些程序希望操作所有三种长度的数据，于是 80386 系统结构提供一种不用明显增加代码长度的方便途径来指定每一种形式。它们认为大多数程序中 16 位或 32 位数据占绝大多数，于是设定一个默认的较长长度是有意义的。这个默认的数据长度由代码段寄存器中的一位指定。若要改变默认数据长度，需在指令前附加 8 位前缀告诉机器这条指令使用其他数据长度。

使用前缀是从 8086 借鉴过来的，8086 可使用多种前缀来改变指令的行为。最初的三个前缀包括忽略默认的段寄存器，给总线加锁来支持同步（见 2.11 节），或重复后面的指令直到寄存器 ECX 减少到 0。最后一个前缀要配合一个字节传送指令使用以便传送可变数目的字节。80386 还加入一个前缀以改变默认的地址长度。

x86 整数操作可以分为 4 个主要的类：

- 1) 数据传送指令，包括 move、push 和 pop。
- 2) 算术和逻辑指令，包括测试、整数和小数算术运算。
- 3) 控制流，包括条件分支、无条件跳转、调用和返回。
- 4) 字符串指令，包括字符串传送和字符串比较。

除了算术和逻辑操作指令的结果既可以保存在寄存器也可以保存在存储器地址外，前两个种类没有值得关注之处。图 2-39 展示了典型的 x86 指令和它们的功能。

x86 的条件分支像 ARMv7 一样基于条件码（condition code）或标志位（flag）。条件码是作为一些操作的副作用被设置的，大部分被用作将结果与 0 比较，然后使用分支指令测试条件码。PC 相对分支地址必须以字节数来指定，这与 ARMv7 和 MIPS 不同，80386 的指令不都是 4 字节长的。

指令	功能
je name	if equal (condition code) {EIP = name}; EIP - 128 <= name < EIP + 128
jmp name	EIP = name
call name	SP = SP - 4; M[SP] = EIP + 5; EIP = name;
movw EBX,[EDI+45]	EBX = M[EDI+45]
push ESI	SP = SP - 4; M[SP] = ESI
pop EDI	EDI = M[SP]; SP = SP + 4
add EAX,#6765	EAX = EAX + 6765
test EDX,#42	Set condition code (flags) with EDX and 42
movs l	M[EDI] = M[ESI]; EDI = EDI + 4; ESI = ESI + 4

图 2-39 x86 的一些典型指令和它们的功能。常用操作的列表在图 2-40 中。CALL 将下一条指令的 EIP 保存在栈上。(EIP 是 Intel 的程序计数器。)

字符串指令是 x86 的祖先 8080 的一部分，在大部分程序中都不使用。它们常常比同等功能的软件例程要慢（见 2.19 节的谬误）。

图 2-40 列出了一些 x86 的整数指令。这些指令大部分都同时有字节和字格式。

指令	含义
控制指令	条件和无条件分支
jnz,jz	条件成立跳转到 EIP + 8 位偏移量；JNE (for JNZ), JE (for JZ) 两者之一
jmp	无条件跳转——8 位或 16 位偏移量
call	过程调用——16 位偏移量；返回地址压入栈中
ret	从栈中弹出返回地址并跳转到该地址处
loop	循环分支——递减 ECX；如果 ECX 非零，则跳转到 EIP + 8 位偏移处
数据传输	在寄存器之间或寄存器和存储器之间传递数据
move	在两个寄存器之间或寄存器和存储器之间传递数据
push,pop	将源操作数压栈；将栈顶数据取到寄存器中
les	从存储器中取 ES 和一个 GPR
算术、逻辑	使用数据寄存器和存储器的算术和逻辑操作
add,sub	将源操作数与目的操作数相加；从目的操作数中减去源操作数；寄存器 - 存储器格式
cmp	比较源和目的操作数；寄存器 - 存储器格式
shl,shr,rcr	左移；逻辑右移；循环右移并用条件码填充
cbw	将 EAX 最右 8 位字节转换成 EAX 最右 16 位字
test	将源操作数和目的操作数进行逻辑与，并设置条件码
inc,dec	递增目的操作数，递减目的操作数
or,xor	逻辑或；异或；寄存器 - 存储器格式
字符串	在字符串操作数之间移动；由重复前缀给出长度
movs	通过递增 ESI 和 EDI 从源字符串复制到目的字符串；可能使用重复
lod\$	从字符串中取字节、字或双字到寄存器 EAX

图 2-40 一些典型的 x86 操作。很多操作使用寄存器 - 存储器格式，这种格式要求源操作数或目的操作数可以是存储器，另一个操作数可以是寄存器或立即数。

2.17.4 x86 指令编码

把最糟的放在最后——80386 的指令编码是非常复杂的，有多种不同指令格式。当没有操作数的时候，80386 的指令可以是 1 字节，最长到 15 字节。

图 2-41 展示了图 2-39 中几条指令的格式。操作码字节中通常有一位用来表明操作数是 8 位还是 32 位。一些指令的操作码可能还包含寻址模式和寄存器，例如，很多的指令具有如下形式“寄存器 = 寄存器操作立即数”。其他指令使用寻址模式的“后置字节”或额外的操作码字节，标记为“mod, reg, r/m”（模式，寄存器，寄存器/存储器）。这个后置字节在寻址存储器的很多指令中都被用到。基址加比例下标的寻址模式需要使用第二个后置字节，标记为“sc, index, base”（比例，下标，基址）。

a.JE EIP + displacement

4	4	8
JE	条件	偏移量

b.CALL

8	32
CALL	位移量

c.MOV EBX, [EDI + 45]

6	1	1	8	8
MOV	d	w	r/m	偏移量

d.PUSH ESI

5	3
PUSH	Reg

e.ADD EAX, #6765

4	3	1	32
ADD	Reg	w	立即数

f.TEST EDX, #42

7	1	8	32
TEST	w	寻址方式字节	立即数

图 2-41 典型的 x86 指令格式。图 2-42 给出后置字节（postbyte）的编码。很多指令包含 1 位的 w 段，这个字段说明操作的是一个字节还是一个双字。MOV 中 d 字段用于从存储器中传出或传入数据的指令并指明传输方向。ADD 指令需要 32 位的立即数字段，因为在 32 位模式下，立即数或者是 8 位或者是 32 位。TEST 中的立即数字段也是 32 位长，是因为在 32 位模式下没有 8 位的立即数要判断。总的来说，指令长度可以从 1 字节到 15 字节变化。较长的长度产生于额外的 1 字节前缀，该长度具有 4 字节的立即数和 4 字节的偏移地址，使用 2 字节的操作码，并使用比例下标模式说明符，这还需要一个额外的字节。

图 2-42 展示了 16 位和 32 位模式的两个后置字节地址指定的编码。不幸的是，为了全面理解哪个寄存器和哪种寻址模式可用，你需要看所有寻址模式的编码，有时甚至需要看指令编码。

reg	w = 0	w = 1	r/m	mod = 0		mod = 1		mod = 2		mod = 3
		16b	32b		16b	32b	16b	32b		
0 AL	AX	EAX	0	addr = BX + SI	= EAX	same	same	same	same	same
1 CL	CX	ECX	1	addr = BX + DI	= ECX	addr as	addr as	addr as	addr as	as
2 DL	DX	EDX	2	addr = BP + SI	= EDX	mod = 0	mod = 0	mod = 0	mod = 0	reg
3 BL	BX	EBX	3	addr = BP + SI	= EBX	+ disp8	+ disp8	+ disp16	+ disp32	field
4 AH	SP	ESP	4	addr = SI	= (sib)	SI + disp8	(sib) + disp8	SI + disp8	(sib) + disp32	"
5 CH	BP	EBP	5	addr = DI	= disp32	DI + disp8	EBP + disp8	DI + disp16	EBP + disp32	"
6 DH	SI	ESI	6	addr = disp16	= ESI	BP + disp8	ESI + disp8	BP + disp16	ESI + disp32	"
7 BH	DI	EDI	7	addr = BX	= EDI	BX + disp8	EDI + disp8	BX + disp16	EDI + disp32	"

图 2-42 x86 的第一个地址说明符的编码：mod，reg，r/m。前 4 列表示 3 位的 reg 字段，它依赖于操作码中的 w 位以及机器是工作在 16 位（8086）模式还是 32 位（80386）模式。余下的列解释了 mod 和 r/m 字段。3 位的 r/m 字段依赖于 2 位的 mod 字段和地址的大小。用于地址计算的寄存器列在第六和第七列中，mod = 0 时依赖于寻址模式，mod = 1 时加上 8 位的偏移量，mod = 2 时加上 16 位或 32 位的偏移量。例外的情况有以下几种：1) 当 mod = 1 或 mod = 2，在 16 位模式时，r/m = 6 选择 BP 加上偏移。2) 当 mod = 1 或 mod = 2，在 32 位模式时，r/m = 5 选择 EBP 加上偏移量。3) 当 mod 不等于 3，在 32 位模式时，r/m = 4，(sib) 代表使用图 2-38 中的比例下标模式。当 mod = 3 时，r/m 字段指定一个寄存器，与 w 位组合在一起和 reg 字段的编码相同。

2.17.5 x86 总结

Intel 的 16 位微处理器比它的竞争对手的更优秀的体系结构（如 Motorola 68000）早两年问世，这个领先使得 IBM 选用 8086 作为其 PC 的 CPU。Intel 的工程师普遍认识到 x86 要比 ARMv7 和 MIPS 的计算机更难制造，但是巨大的市场意味着 AMD 和 Intel 可以投入更多的资源来克服这些额外的复杂性。数量上的巨大优势弥补了风格上的缺点，这使得 x86 前景美好。

x86 中最常使用的体系结构组成部分是不难实现的，从 1978 年开始 AMD 和 Intel 就展示了整数程序性能的快速改进。为了获得这样的性能，编译器必须避免那些难于实现快速执行的体系结构部分。

然而，在后 PC 时代，虽然有大量的体系结构和制造专家基于 x86 做工作，但是 x86 在个人移动设备里面还不具有竞争力。

2.18 实例：ARMv8（64 位）指令集

在一个指令集所具有的所有潜在问题中，最不可能解决的就是地址空间太小的问题。x86 是第一个扩展为 32 位地址的，并且是第一个扩展为 64 位地址的指令集，许多其他的指令系统都被落在了后面。例如，虽然具有 16 位地址的 MOStek 6502 指令集统治了 Apple II，但是 Apple II 即使是第一个成功的商用个人计算机的领头羊，却也由于其地址空间上的缺陷饱受诟病。

虽然 ARM 体系结构遇到了 32 位地址空间的限制，在 2007 年开始设计具有 64 位地址的 ARM，并最终在 2013 年完成。与 x86 中为了使寄存器加宽为 64 位只做了很小改变不同，ARM 作了完全的改进。如果你了解 MIPS，那就非常容易了解 64 位版本的 ARMv8。

首先，与 MIPS 相比，ARM 舍弃了 v7 中并不常用的一些特性：

- v8 中没有条件执行字段，而在 v7 中几乎每条指令都有该字段。
- 立即数字段仅仅是一个 12 位的常数，而在 v7 中是产生一个常数的函数的输入。
- ARM 舍弃了 Load Multiple 和 Store Multiple 指令。
- PC 不再是一个寄存器，因此如果对其进行写操作将会导致非预期的分支转移。

其次，ARM 添加了一些 MIPS 中有用的特征：

- v8 有 32 个通用寄存器，编译器设计者非常喜欢该特点。与 MIPS 相同，一个寄存器永远存放 0，虽然在 load 和 store 指令中该寄存器将由栈指针替代。
- ARMv8 的寻址方式是用于所有的字长，而在 ARMv7 中并非如此。
- 它包含了 ARMv7 中省掉的除法指令。
- 它增加了 MIPS 中的相等或不等的条件分支指令。

由于 v8 相对于 v7 而言，其指令集更像 MIPS，因此我们的结论是 ARMv7 和 ARMv8 的主要相同点仅仅是名字。

2.19 谬误与陷阱

谬误：更强大的指令意味着更高的性能。

x86 的一个强大的地方是可以通过前缀来改变后续指令的执行。某个前缀可以重复执行后面的指令直到一个计数器减少至 0。因此，为了在存储器中传输数据，看起来最自然的指令序列应该是使用加了重复前缀的 move 指令来实现 32 位的存储器到存储器的传输。

另外一种方法是使用所有计算机上都有的标准指令，将数据取到寄存器后再存回存储器。这种形式通过代码复制来减少循环开销，复制操作大约快 1.5 倍。第三种方式，使用更大的浮点寄存器代替 x86 的整数寄存器，复制操作比使用复杂指令快 2 倍。

谬误：使用汇编语言编程来获得最高的性能。

在一段时间内，编程语言的编译器经常产生很低级的指令序列。通过不断改进，编译器产生的代码与手工编写的代码在性能上的差距正在快速缩小。事实上，为了与当今编译器竞争，汇编程序员需要深刻理解第 4 章和第 5 章中的计算机体系结构概念（包括处理器流水线和存储器层次）。

编译器和汇编程序员之间的斗争正在逐渐消失。例如，C 为程序员提供一个指示编译器把变量保存在寄存器中而不是换出到存储器中的机会。当编译器在寄存器分配上能力较差时，这种指示对性能至关重要。事实上，一些较老的 C 语言课本花费大量的时间给出了有效的寄存器指示的例子。今天的 C 语言编译器通常忽略这种指示，因为编译器能比程序员更好地分配寄存器。

即使手工编写会产生更快的代码，汇编语言编写还是存在很多危险：需要更多时间编码和调试，可移植性差，难于维护。软件工程中少数几个被广泛接受的公理之一是编写的程序行数越多所花时间也越多。很明显使用汇编语言编写的程序比 C 或 Java 更长。一旦代码写好，下一个危险将是它会变成一个流行的程序。这种程序存在的时间总是比预期要长，意味着程序员需要每隔几年就更新一下代码使新的版本可以运行在新的操作系统和新机器上。高级语言而不是汇编语言编写的程序不仅可以使未来的编译器为未来的机器生成代码，还可以使软件易于维护并运行在其他类型的计算机上。

谬误：商用计算机二进制兼容的重要性意味着成功的指令集不需改变。

在向后的二进制兼容是神圣不可侵犯的同时，图 2-43 显示了 x86 指令集的快速发展。在 35 年中，平均每个月至少增加一条新的指令。

陷阱：忘记在字节寻址的机器中，连续的字地址相差不是 1。

很多汇编程序员假定下一个字地址可以通过将寄存器的值加 1 来获得，而不是增加一个字的字节数，这使他们犯下很多错误。提前注意以便有所准备！

陷阱：在自动变量的定义过程外，使用指针指向该变量。

处理指针的常见错误是使用指向一个过程中局部数组的指针，从该过程传出结果。遵从图 2-12 中的栈规则，当过程返回时，包含局部数组的存储器将立即被重新使用。指向自动变量的指针会造成混乱。

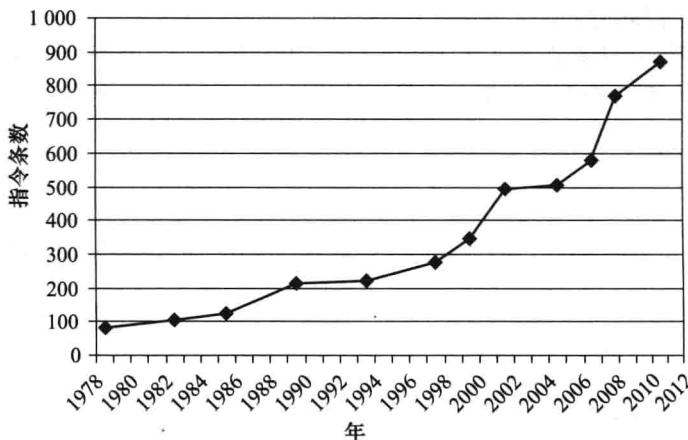


图 2-43 随时间推移 x86 指令集的增长。这种扩展是有一定的技术价值的，迅速的变化也增加了其他公司试图做兼容处理器的难度

2.20 本章小结

少就是多。

——Robert Browning, 《Andrea del Sarto》, 1855

存储程序计算机的两个准则是指令的使用与数字没有区别，以及使用可修改的存储器。这些准则使一台计算机可以在不同的领域辅助环境科学家、经济顾问和小说家。选择机器可以理解的指令集需要精妙的平衡程序执行需要的指令数目、指令执行所需的时钟周期数和时钟的速度。就像本章所描述的，在做精妙平衡时有 3 条准则可以指导设计者：

1) 简单源于规整。规整性使 MIPS 指令集具有很多特点：所有指令长度统一、算术指令总是需要三个寄存器操作数和寄存器字段在每种指令格式的位置相同。

2) 越小越快。对速度的要求导致 MIPS 只有 32 个寄存器而不是更多。

3) 优秀的设计需要好的折中。一个 MIPS 例子是在指令中提供更大地址与常数，并且保持所有的指令具有相同的长度之间的折中。

与计算机体系结构相同，在指令集中也会看到“加速大概率事件”的伟大思想。该思想在 MIPS 中的体现包括条件分支的 PC 相对寻址和大常数操作数的立即数寻址。

机器语言之上是人们可读的汇编语言。汇编器将翻译为机器可以理解的二进制数，它甚至通过创造硬件中没有的符号指令来“扩展”指令集。例如，较大的常量和地址被切割成合适的大小，常用的指令变体都有它们自己的名字，等等。图 2-44 列举了到目前为止我们讲过的 MIPS 指令，包括实际指令和伪指令。在更高级别隐藏细节是伟大思想“抽象”的另外一个例子。

每一类 MIPS 指令与编程语言中出现的结构相关：

- 算术指令对应于赋值语句中的运算。
- 传输指令很可能发生在处理像数组和结构体这样的数据结构时。
- 条件分支被用于 if 语句和循环。
- 无条件分支被用于过程调用和返回以及 case/switch 语句。

这些指令出现频率不相等，少数指令出现频率较大。例如，图 2-45 展示了 SPEC CPU 2006 中每类指令出现的频率。指令出现频率的不同在数据通路、控制通路和流水线的特征分析中扮演重要角色。

MIPS 指令	名称	格式	MIPS 伪指令	名称	格式
加	add	R	移位	move	R
减	sub	R	乘	mult	R
加立即数	addi	I	乘立即数	multi	I
取字	lw	I	取立即数	li	I
存字	sw	I	小于时跳转	blt	I
取半字	lh	I	小于或等于时跳转	ble	I
取无符号半字	lhu	I	大于时跳转	bgt	I
存半字	sh	I	大于或相等时跳转	bge	I
取字节	lb	I			
取无符号字节	lbu	I			
存字节	sb	I			
取链接字	ll	I			
存条件字	sc	I			
取立即数高位	lui	I			
与	and	R			
或	or	R			
或非	nor	R			
与立即数	andi	I			
或立即数	ori	I			
逻辑左移	sll	R			
逻辑右移	srl	R			
相等时跳转	bne	I			
不相等时跳转	bne	I			
小于时置位	slt	R			
小于立即数时置位	slti	I			
小于无符号立即数时置位	sltiu	I			
跳转	j	J			
跳转至寄存器所指位置	jr	R			
跳转和链接	jal	J			

图 2-44 到目前为止介绍过的 MIPS 指令集，左侧是真实的 MIPS 指令，右侧是伪指令。附录 A (A.10 节) 描述了完整的 MIPS 体系结构。图 2-1 展示了与本章相关的更细致的 MIPS 体系结构。这里给出的信息可在 MIPS 参考数据卡的第 1 和第 2 列查到

指令类别	MIPS 范例	相应的高级语言	出现频率	
			整型	浮点
算术	add, sub, addi	赋值语句中的操作	16%	48%
数据传输	lw, sw, lb, lbu, lh, lhu, sb, lui	对数据结构的引用，例如数组	35%	36%
逻辑	and, or, nor, andi, ori, sll, srl	赋值语句中的操作	12%	4%
条件分支	beq, bne, slt, slti, sltiu	if 语句和循环	34%	8%
跳转	j, jr, jar	过程调用，返回，case/switch 语句	2%	0%

图 2-45 MIPS 指令分类、范例以及相应的高级编程语言结构和 SPEC CPU 2006 测试程序执行时定、浮点指令所占的比例。第 3 章中的图 3-26 展示了每条 MIPS 指令执行时所占的平均比例

在第 3 章解释计算机算术运算之后，我们将继续揭示 MIPS 指令集体系结构。

2.21 历史观点和拓展阅读

本节概述了指令集体系结构 (ISA) 的历史，我们介绍了编程语言和编译器的简短历史。

ISA 包括累加器体系结构、通用寄存器体系结构、栈体系结构和 ARM 及 x86 的简史。我们还回顾了高级语言计算机体系结构中的争议问题和精简指令集体体系结构。编程语言的历史包括 Fortran、Lisp、Algol、C、Cobol、Pascal、Simula、Smalltalk、C++ 和 Java。编译器的历史包括重要的里程碑和实现它们的先驱。本节剩余部分在配套网站中的 2.21 节中。

2.22 练习题

附录 A 描述了对这些练习有帮助的 MIPS 的模拟器。尽管模拟器可以接受伪指令，但是在要求产生 MIPS 代码的习题中，尽量不要使用伪指令。你的目的是学习实际的 MIPS 指令集，如果问你指令数，你所给出的答案必须反映实际执行的指令数而不是伪指令。

有些情况必须使用伪指令（例如，当汇编时不知道真实值时，使用 `la` 指令）。还有些情况下，使用伪指令会更方便并使代码可读性变好（例如，`li` 和 `move` 指令）。如果你因为这些原因选择使用伪指令，请在伪指令开始的地方加上一两句话，说明你使用伪指令的原因。

2.1 [5] <2.2> 下面的 C 语言表达式对应的 MIPS 汇编语言代码是什么？假设给定变量 `f`、`g`、`h` 和 `i`，像在 C 程序中声明的一样它们都是 32 位的整数，使用最少的 MIPS 汇编指令。
`f = g + (h - 5);`

2.2 [5] <2.2> 下面的 MIPS 汇编语言程序段对应的 C 语言表达式是什么？

```
add  f, g, h
add  f, i, f
```

164

2.3 [5] <2.2, 2.3> 下面的 C 语言表达式对应的 MIPS 汇编代码是什么？假设变量 `f`、`g`、`h`、`i` 和 `j` 分别赋值给寄存器 `$s0`、`$s1`、`$s2`、`$s3` 和 `$s4`。假设数组 `A` 和 `B` 的基址分别在寄存器 `$s6` 和 `$s7` 中。

```
B[8] = A[i-j];
```

2.4 [5] <2.2, 2.3> 下面的 MIPS 汇编语言程序段对应的 C 语言表达式是什么？假设变量 `f`、`g`、`h`、`i` 和 `j` 分别赋值给寄存器 `$s0`、`$s1`、`$s2`、`$s3` 和 `$s4`。假设数组 `A` 和 `B` 的基址分别在寄存器 `$s6` 和 `$s7` 中。

```
sll  $t0, $s0, 2      # $t0 = f * 4
add  $t0, $s6, $t0    # $t0 = &A[f]
sll  $t1, $s1, 2      # $t1 = g * 4
add  $t1, $s7, $t1    # $t1 = &B[g]
lw   $s0, 0($t0)      # f = A[f]
addi $t2, $t0, 4
lw   $t0, 0($t2)
add  $t0, $t0, $s0
sw   $t0, 0($t1)
```

2.5 [5] <2.2, 2.3> 在不改变功能的前提下，重写习题 2.4 中的 MIPS 程序使其指令数目尽可能少。

2.6 下表表示在主存中存放的一个数组的 32 位数据。

Address	Data
24	2
38	4
32	3
36	6
40	1

165

2.6.1 [5] <2.2, 2.3> 基于上表中数据在存储器中的位置，编写一段 C 代码，将数据从小到大排序，最小的数放在地址最低的位置。（假设这段数据代表了 C 的一个 Int 型数组 `Array`，并且这台特别的机器是按照字节寻址的，且一个字包含 4 字节。）

2.6.2 [5] <2.2, 2.3> 基于上表中数据在存储器中的位置，编写一段 MIPS 代码，将数据从小到大排序，最小的数放在地址最低的位置。（使用最少的 MIPS 汇编指令，假设 `Array` 的基址保存在寄存器 `$s6` 中。）

2.7 [5] <2.3> 分别画出数据 0xabcdef12 在大端编址和小端编址的机器上是如何分布在存储器中的。(假定数据从地址 0 开始存储。)

2.8 [5] <2.4> 将 0xabcdef12 转化为十进制。

2.9 [5] <2.2, 2.3> 把下面的 C 代码翻译为 MIPS 代码。假定变量 f、g、h、i 和 j 分别赋值给寄存器 \$s0、\$s1、\$s2、\$s3 和 \$s4。假定数组 A 和数组 B 的基址分别存放在 \$s6 和 \$s7 中。假定数组 A 和数组 B 中的元素均为 4 字节的字：

B[8] = A[i] + A[j];

2.10 [5] <2.2, 2.3> 把下面的 MIPS 代码翻译为 C 代码。假定变量 f、g、h、i 和 j 分别赋值给寄存器 \$s0、\$s1、\$s2、\$s3 和 \$s4。假定数组 A 和数组 B 的基址分别存放在 \$s6 和 \$s7 中。

```
addi $t0, $s6, 4
add $t1, $s6, $0
sw $t1, 0($t0)
lw $t0, 0($t0)
add $s0, $t1, $t0
```

2.11 [5] <2.2, 2.5> 对于每条 MIPS 指令，写出操作码 (OP)、源操作数 (RS) 和目标操作数 (RT) 的值 (value)。对于 I 型指令，写出立即数字段的值。对于 R 型指令，写出目的寄存器 (RD) 字段的值。

2.12 假定寄存器 \$s0 和 \$s1 分别存放数值 0x80000000 和 0xD0000000。

2.12.1 [5] <2.4> 下面汇编代码的 \$t0 的值是多少？

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

2.12.2 [5] <2.4> \$t0 中的结果是期望的结果还是发生溢出后的结果？

2.12.3 [5] <2.4> 对于上面定义的寄存器 \$s0 和 \$s1 的内容，下面汇编代码的 \$t0 的值是多少？

sub \$t0, \$s0, \$s1

2.12.4 [5] <2.4> \$t0 中的结果是期望的结果还是发生溢出后的结果？

2.12.5 [5] <2.4> 对于上面定义的寄存器 \$s0 和 \$s1 的内容，下面汇编代码的 \$t0 的值是多少？

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

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

2.12.6 [5] <2.4> \$t0 中的结果是期望的结果还是发生溢出后的结果？

2.13 假定 \$s0 中的值为 128_{10} 。

2.13.1 [5] <2.4> 对于指令 add \$t0, \$s0, \$s1，求使结果产生溢出的 \$s1 的值的范围。

2.13.2 [5] <2.4> 对于指令 sub \$t0, \$s0, \$s1，求使结果产生溢出的 \$s1 的值的范围。

2.13.3 [5] <2.4> 对于指令 sub \$t0, \$s1, \$s0，求使结果产生溢出的 \$s1 的值的范围。

2.14 [5] <2.2, 2.5> 写出下面的二进制数值对应的类型和汇编语言指令：

0000 0010 0001 0000 1000 0000 0010 0000₂

2.15 [5] <2.2, 2.5> 给出下面指令的类型和十六进制表示：sw \$t1, 32 (\$t2)

2.16 [5] <2.5> 写出用下面 MIPS 字段描述的指令的类型、汇编语言指令和二进制表示：

op=0, rs=3, rt=2, rd=3, shamt=0, funct=34

2.17 [5] <2.5> 写出用下面 MIPS 字段描述的指令的类型、汇编语言指令和二进制表示：

op=0x23, rs=1, rt=2, const=0x4

2.18 假设可以将 MIPS 寄存器文件扩展到 128 个寄存器，并将指令集中的指令数扩展为原来的 4 倍。

2.18.1 [5] <2.5> 这将如何影响 R 型指令的每个位字段的大小？

2.18.2 [5] <2.5> 这将如何影响 I 型指令的每个位字段的大小？

2.18.3 [5] <2.5, 2.10> 在提出的这两种变化中，每种变化如何减少一个 MIPS 汇编程序的大小？另一方面，如何增大一个 MIPS 汇编程序的大小？

2.19 假设如下寄存器内容：

\$t0 = 0xAAAAAAAA, \$t1 = 0x12345678

2.19.1 [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 \$t2 的值是多少？

166

167

```
sll $t2, $t0, 44
or $t2, $t2, $t1
```

- 2.19.2** [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 \$t2 的值是多少？

```
sll $t2, $t0, 4
andi $t2, $t2, -1
```

- 2.19.3** [5] <2.6> 对于以上的寄存器内容，执行下面的指令序列后 \$t2 的值是多少？

```
srl $t2, $t0, 3
andi $t2, $t2, 0xFFFF
```

168

- 2.20** [5] <2.6> 找出完成如下功能的最短的 MIPS 指令序列：从寄存器 \$t0 中提取第 16 位到第 11 位，然后使用这些位替换寄存器 \$t1 的第 31 位到第 26 位，保持其他位不变。

- 2.21** [5] <2.6> 写出可用来实现下面伪指令的 MIPS 指令集的最小子集：

```
not $t1, $t2 // bit-wise invert
```

- 2.22** [5] <2.6> 对于下面的 C 语言表达式，写一个能够完成同样操作的最短 MIPS 汇编指令程序段。假设 \$t1 = A, \$t2 = B, \$s1 是 C 的基地址。

```
A = C[0] << 4;
```

- 2.23** [5] <2.7> 假设 \$t0 中存放数值 0x00101000，在执行下列指令后 \$t2 的值是多少？

```
slt $t2, $0, $t0
bne $t2, $0, ELSE
j DONE
ELSE: addi $t2, $t2, 2
DONE:
```

- 2.24** [5] <2.7> 假设程序计数器 (PC) 被设置为 0x2000 0000，是否可以使用 MIPS 的跳转 (j) 指令将 PC 设置为地址 0x4000 0000？是否可以使用 MIPS 的相等则分支 (beq) 指令将 PC 设置为该地址？

- 2.25** MIPS 指令集不包含下面的指令：

```
rpt $t2, loop # if(R[rs]>0) R[rs]=R[rs]-1, PC=PC+4+BranchAddr
```

- 2.25.1** [5] <2.7> 如果要在 MIPS 指令集中实现该指令，哪种指令格式最合适？

- 2.25.2** [5] <2.7> 能够实现相同操作的最短 MIPS 指令序列是什么？

- 2.26** 考虑如下的 MIPS 循环：

```
LOOP: slt $t2, $0, $t1
      beq $t2, $0, DONE
      subi $t1, $t1, 1
      addi $s2, $s2, 2
      j LOOP
DONE:
```

- 2.26.1** [5] <2.7> 假设寄存器 \$t1 的初始值为 10，假设 \$t2 初始值为 0，则循环执行完毕时寄存器 \$t2 的值是多少？

- 2.26.2** [5] <2.7> 对于上面的循环体，写出等价的 C 代码例程。假定寄存器 \$s1、\$s2、\$t1 和 \$t2 分别为整数 A、B、i 和 temp。

- 2.26.3** [5] <2.7> 假定寄存器 \$t1 的初始值为 N，上面的 MIPS 汇编循环执行了多少条指令？

- 2.27** [5] <2.7> 将下面的 C 代码翻译为 MIPS 汇编代码。要求使用的指令数目最少。假设值 a、b、i 和 j 分别存放在寄存器 \$s0、\$s1、\$t0 和 \$t1 中。另外假设寄存器 \$s2 中存放着数组 D 的基地址。

```
for(i=0; i<a; i++)
    for(j=0; j<b; j++)
        D[4*j] = i + j;
```

- 2.28** [5] <2.7> 实现习题 2.27 中的 C 代码用了多少条 MIPS 汇编指令？如果变量 a 和 b 分别初始化为 10 和 1，并且 D 中所有元素初始化为 0，将整个循环执行完成时，一共执行了多少条 MIPS 指令？

- 2.29** [5] <2.7> 将下面的循环翻译成 C 代码。假定寄存器 \$t1 中存放 C 语言级的整数 i，\$s2 中存放 C 语言级的整数 result，\$s0 存放整数数组 MemArray 的基地址。

169

```

    addi $t1, $0, $0
LOOP: lw    $s1, 0($s0)
      add  $s2, $s2, $s1
      addi $s0, $s0, 4
      addi $t1, $t1, 1
      slti $t2, $t1, 100
      bne  $t2, $s0, LOOP

```

170

- 2.30** [5] <2.7> 将习题 2.29 中的循环重写以减少执行的 MIPS 指令。
- 2.31** [5] <2.8> 使用 MIPS 汇编实现下面的 C 代码。该函数一共执行了多少条 MIPS 指令？
- ```

int fib(int n){
 if (n==0)
 return 0;
 else if (n == 1)
 return 1;
 else
 return fib(n-1) + fib(n-2);
}

```
- 2.32** [5] <2.8> 函数经常被编译器实现为内联“in-line”的形式。内联函数是将函数体复制到程序空间中，以消除函数调用的开销。对于上面的函数，请用 MIPS 汇编实现内联版本的 C 代码。请问实现这个函数总共可以减少多少条 MIPS 汇编指令？（假设 C 的变量 n 被初始化为 5。）
- 2.33** [5] <2.8> 对于每一次函数调用，画出调用后栈的内容。（假定栈指针被初始化为 0x7fffffc，寄存器的使用情况和图 2-11 相同。）
- 2.34** 将下面的函数翻译成 MIPS 汇编语言。如果需要使用寄存器 \$t0 到 \$t7，请从编号小的寄存器开始使用。假设函数 func 的声明为 “int f (int a, int b);”，函数 f 的代码如下：
- ```

int f(int a, int b, int c, int d){
    return func(func(a,b),c+d);
}

```
- 171
- 2.35** [5] <2.8> 请问这个函数可以使用尾调用优化吗？如果不能，请说明原因。如果能，请说明优化前后执行 f 的指令数的差别。
- 2.36** [5] <2.8> 在习题 2.34 中函数 f 返回之前，我们可以知道寄存器 \$t5、\$s3、\$ra 和 \$sp 的内容吗？（注意，我们知道函数 f 的全部，但是我们只知道函数 func 的声明。）
- 2.37** [5] <2.9> 用 MIPS 汇编语言写一段代码将包含十进制正整数和负整数的 ASCII 码的数串转换成整数。在程序中使用寄存器 \$a0 处理由数字 0~9 组成的非空串的地址。程序应该计算与这个数字串等值的整数，并将这个整数存放在寄存器 \$v0 中。如果在字符串的任意位置出现非数字字符，程序停止并将 -1 存入 \$v0。例如，如果寄存器 \$a0 指向 3 字节的序列 50₁₀, 52₁₀, 0₁₀（非终结的字符串“24”），当程序停止的时候，寄存器 \$v0 中的值应该是 24₁₀。
- 2.38** [5] <2.9> 对于如下代码：
- ```

lbu $t0, 0($t1)
sw $t0, 0($t2)

```
- 假设寄存器 \$t1 中存放地址 0x1000 0000，寄存器 \$t2 中存放地址 0x1000 0010。注意 MIPS 体系结构使用大端地址。假设地址 0x1000 0000 的数据是 0x11223344。寄存器 \$t2 指向的地址中存放的数值是多少？
- 2.39** [5] <2.10> 请编写能产生 32 位常数 0010 0000 0000 0001 0100 1001 0010 0100<sub>2</sub> 的 MIPS 代码，并将值存储到寄存器 \$t1 中。
- 2.40** [5] <2.6, 2.10> 如果当前 PC 值是 0x00000000，可以使用单独的跳转指令跳转到练习题 2.39 中所指定的 PC 地址吗？
- 2.41** [5] <2.6, 2.10> 如果当前 PC 值是 0x00000600，可以使用单独的分支指令跳转到练习题 2.39 中所指定的 PC 地址吗？
- 2.42** [5] <2.6, 2.10> 如果当前 PC 值是 0x1FFF f000，可以使用单独的分支指令跳转到练习题 2.39 中所指定的 PC 地址吗？
- 172

- 2.43** [5] <2.11> 写出实现下面 C 代码的 MIPS 汇编代码:

```
lock(1k);
shvar=max(shvar,x);
unlock(1k);
```

假设变量 1k 的地址在 \$a0 中, 变量 shvar 的地址在 \$a1 中, 变量 x 的地址在 \$a2 中。你所编写的这个重要部分的代码不能包含任何函数调用。使用 ll/sc 指令实现 lock() 操作, 而 unlock() 操作可以简单地使用存数指令。

- 2.44** [5] <2.11> 重新解决练习题 2.43 中的问题, 不过这次使用 ll/sc 直接完成 shvar 变量的原子更新操作, 不使用 lock() 和 unlock()。注意这个问题中没有变量 1k。

- 2.45** [5] <2.11> 以练习题 2.43 中的代码为例, 解释当两个处理器同时执行这段临界区域时, 将发生什么情况? 假设每个处理器执行一条指令正好需要一个周期。

- 2.46** 假设给定处理器的算术指令的 CPI 是 1, 取数/存数指令的 CPI 是 10, 分支指令的 CPI 是 3。假设一个程序由 5 亿条算术指令、3 亿条取数/存数指令和 1 亿条分支指令组成。

- 2.46.1** [5] <2.19> 假设向指令集中添加了新的、功能更强的算术指令。通过使用这些功能更强大的算术指令平均可以减少程序执行所需要的 25% 的算术指令, 而时钟周期的开销增长了 10%。请问这是好的设计选择吗? 为什么?

- 2.46.2** [5] <2.19> 假设我们找到一种可以使算术指令性能达到原来两倍的方法。请问我们机器的整体加速是多少? 假设我们找到一种可以使算术指令性能达到原来 10 倍的方法, 那么机器的性能整体加速又是多少?

- 2.47** 假设一给定程序共执行了 70% 的算术指令、10% 的取数/存数指令和 20% 的分支指令。

- 2.47.1** [5] <2.19> 假设执行一条算术指令、取数/存数指令和分支指令分别需要 2 个周期、6 个周期和 3 个周期, 求平均 CPI。

- 2.47.2** [5] <2.19> 在取数/存数指令和分支指令执行时间不变的情况下, 如果要使性能提升 25%, 则算术运算指令的平均执行时间应该为多少?

- 2.47.3** [5] <2.19> 在取数/存数指令和分支指令执行时间不变的情况下, 如果要使性能提升 50%, 则算术运算指令的平均执行时间应该为多少?

## 01 小测验答案

**2.2** MIPS, C, Java

**2.3** 2. 非常慢

**2.4** 2.  $-8_{10}$

**2.5** 4. sub \$t2, \$t0, \$t1

**2.6** 都可以。将“逻辑与”和全“1”的掩码一起使用会导致除了想要的区域之外, 都变成 0。正确的左移位操作将左边的位数都移走。合适的右移将一个字最右边的区域都移走, 将 0 留在字中。注意到“逻辑与”操作会保留原始的值, 移位操作对将需要的区域移动到字的最右边。

**2.7** I. 全对, II. 1。

**2.8** 两个都正确。

**2.9** I. 1 和 2, II. 3。

**2.10** I. 4. + -128K, II. 6. 一个 256M 的块, III. 4. sll。

**2.11** 两个都正确。

**2.12** 4. 与机器无关。