然后用这两个指令将参数传递到 swap：

```
mv x10, x21          // first swap parameter is v
mv x11, x20          // second swap parameter is j
```

保留 sort 中的寄存器

唯一剩下的代码是寄存器的保存和恢复。显然，我们必须在寄存器 x1 中保存返回地址，因为 sort 是一个过程并且调用自己。sort 过程还使用被调用者保存的寄存器 x19、x20、x21 和 x22，因此必须保存它们。所以 sort 的过程头如下：

```
addi    sp, sp, -40      // make room on stack for 5 regs
sd      x1, 32(sp)       // save x1 on stack
sd      x22, 24(sp)      // save x22 on stack
sd      x21, 16(sp)      // save x21 on stack
sd      x20, 8(sp)       // save x20 on stack
sd      x19, 0(sp)       // save x19 on stack
```

过程尾简单地反转所有这些指令，然后加一个 jalr 以便返回。

完整的 sort 过程

现在我们将所有部分放在一起，如图 2-25 所示，注意在 for 循环中用寄存器 x21 和 x22 替换对寄存器 x10 和 x11 的引用。再一次，为了使代码更容易理解，我们用过程中的每个代码块的用途来标识它们。在此示例中，C 中的 9 行 sort 过程在 RISC-V 汇编语言中变为 34 行。

保存寄存器

```
sort:   addi sp, sp, -40      # make room on stack for 5 registers
        sd x1, 32(sp)         # save return address on stack
        sd x22, 24(sp)        # save x22 on stack
        sd x21, 16(sp)        # save x21 on stack
        sd x20, 8(sp)         # save x20 on stack
        sd x19, 0(sp)         # save x19 on stack
```

过程主体

移动参数	`mv x21, x10` # copy parameter x10 into x21 `mv x22, x11` # copy parameter x11 into x22
外循环	`li x19,0` # i = 0 `for1tst:bge x19, x22, exit1` # go to exit1 if i >= n
内循环	`addi x20, x19, -1` # j = i - 1 `for2tst:blt x20, x0, exit2` # go to exit2 if j < 0 `slli x5, x20, 3` # x5 = j * 8 `add x5, x21, x5` # x5 = v + (j * 8) `ld x6, 0(x5)` # x6 = v[j] `ld x7, 8(x5)` # x7 = v[j + 1] `ble x6, x7, exit2` # go to exit2 if x6 < x7
参数传递和调用	`mv x10, x21` # first swap parameter is v `mv x11, x20` # second swap parameter is j `jal x1, swap` # call swap
内循环	`addi x20, x20, -1` j for2tst `j for2tst` # go to for2tst
外循环	`exit2: addi x19, x19, 1` # i += 1 `j for1tst` # go to for1tst

图 2-25　图 2-24 的 sort 过程的 RISC-V 汇编代码

```
exit1:  ld x19, 0(sp)          # restore x19 from stack
        ld x20, 8(sp)          # restore x20 from stack
        ld x21, 16(sp)         # restore x21 from stack
        ld x22, 24(sp)         # restore x22 from stack
        ld x1, 32(sp)          # restore return address from stack
        addi sp, sp, 40        # restore stack pointer
```

```
        jalr x0, 0(x1)         # return to calling routine
```

图 2-25 （续）

详细阐述 这个例子中一个有效的优化是过程内联。在调用 swap 过程的代码出现的地方，编译器将从 swap 过程体中复制代码，而不是传递参数和使用 jal 指令调用代码。在这个例子中，内联将避免使用四条指令。内联优化的缺点是如果从多个地方调用内联过程，编译后的代码会增大。如果因此增加了 cache 失效率，这样的代码扩展可能导致性能较低；参见第 5 章。

理解程序性能 图 2-26 显示了编译器优化对 sort 程序性能、编译时间、时钟周期、指令数和 CPI 的影响。请注意，未经优化的代码具有最佳的 CPI，而 O1 优化具有最少的指令数，但 O3 是最快的，这说明执行时间是程序性能的唯一准确度量。

无优化	1.00	158 615	114 938	1.38
01（中级）	2.37	66 990	37 470	1.79
02（完全）	2.38	66 521	39 993	1.66
03（过程集成）	2.41	65 747	44 993	1.46

图 2-26 冒泡程序使用编译器优化后的性能、指令数和 CPI 的比较。程序将数组初始化为随机值，对 100 000 个 32 位字进行排序。这些程序在奔腾 4 上运行，时钟频率为 3.06GHz，系统总线为 533MHz，内存大小为 2GB 的 PC2100 DDR SDRAM。操作系统版本为 Linux 版本 2.4.20

图 2-27 比较了编程语言、编译与解释以及算法对各种性能的影响。第 4 列显示对于冒泡排序来说，未优化的 C 程序比解释型的 Java 代码快 8.3 倍。使用 JIT 编译器使 Java 比未优化的 C 语言快 2.1 倍，并且比使用最高优化的 C 代码也就慢不到 1.13 倍。（2.15 节给出了有关 Java 的解释与编译以及冒泡的 Java 和 jalr 代码的更多细节。）第 5 列中快速排序的比率并不接近，可能是因为在较短的执行时间内分摊运行时编译的成本更难。最后一列演示了更好的算法带来的影响，当排序 100 000 个元素时，带来了三个数量级的性能提升。即使将第 5 列中的解释型 Java 与第 4 列中最高优化的 C 代码进行比较，快速排序也会比冒泡排序快 50 倍（0.05×2468 或 123/2.41）。

语言	执行模式	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	无优化	1.00	1.00	2468
	编译器	01	2.37	1.50	1562
	编译器	02	2.38	1.50	1555
	编译器	03	2.41	1.91	1955
Java	解释器	–	0.12	0.05	1050
	即时编译器	–	2.13	0.29	338

图 2-27　两个排序算法的性能，分别使用 C 和 Java，以及分别采用解释和优化编译器相对于未优化的 C 版本的对比。最后一列显示了针对每种语言和执行选项的快速排序相对于冒泡排序的性能优势。这些程序在与图 2-26 相同的系统上运行。JVM 版本是 Sun 1.3.1，JIT 版本是 Sun Hotspot 1.3.1

2.14　数组与指针

理解指针对于所有 C 程序员新手来说都是具有挑战的。通过比较使用数组和数组下标的汇编代码与使用指针的汇编代码，可以从本质上理解指针。本节展示了清除内存中一个双字序列的两个过程的 C 和 RISC-V 汇编版本：一个使用数组下标，另一个使用指针。图 2-28 给出了这两个 C 过程。

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

图 2-28　将一个数组全部清零的两个 C 过程。clear1 使用下标，而 clear2 使用指针。对于不熟悉 C 的人，第二个过程需要做一些解释。变量的地址用 & 表示，而指针指向的对象用 * 表示。声明部分则声明了 array 和 p 是整数的指针。clear2 中 for 循环的第一部分将 array 的第一个元素的地址分配给指针 p。for 循环的第二部分判断指针是否指向超出 array 的最后一个元素。在 for 循环的底部将指针递增 1，意味着将指针移动到其声明大小的下一个顺序对象。由于 p 是指向整数的指针，编译器会生成 RISC-V 指令将 p 递增 8，即 RISC-V 整数中的字节数。在循环中将 0 赋给 p 指向的对象

本节的目的是展示指针如何映射到 RISC-V 指令，而不是支持一种过时的编程风格。在本节末尾，将看到现代编译优化对这两个过程的影响。

2.14.1　用数组实现 clear

首先从数组版本的 clear1 开始，重点关注循环体并忽略过程链接代码。假设两个参数 array 和 size 分别在寄存器 x10 和 x11 中，并且给 i 分配寄存器 x5。

for 循环的第一部分，i 的初始化是很简单的：

```
li    x5, 0    // i = 0 (register x5 = 0)
```

要将 array[i] 设置为 0，必须首先获取其地址。先将 i 乘以 8 得到字节地址：

```
loop1: slli    x6, x5, 3    // x6 = i * 8
```

由于数组的起始地址在寄存器中，必须使用加法指令将其加到下标中以获取 array[i] 的地址：

```
add  x7, x10, x6    // x7 = address of array[i]
```

最后，即可将 0 存到该地址：

```
sd  x0, 0(x7)    // array[i] = 0
```

这条指令是循环体的结尾，因此下一步是增加 i 值：

```
addi x5, x5, 1    // i = i + 1
```

循环测试检测 i 是否小于 size：

```
blt  x5, x11, loop1 // if (i < size) go to loop1
```

现在已经得到了程序的所有片段。以下是使用下标对数组清零的 RISC-V 代码：

```
        li     x5, 0           // i = 0
loop1: slli    x6, x5, 3       // x6 = i * 8
        add    x7, x10, x6     // x7 = address of array[i]
        sd     x0, 0(x7)       // array[i] = 0
        addi   x5, x5, 1       // i = i + 1
        blt    x5, x11, loop1  // if (i < size) go to loop1
```

（只要 size 大于 0，此代码就能正确工作；ANSI C 需要在循环之前测试 size，但我们将在此处跳过该合法性。）

2.14.2 用指针实现 clear

使用指针的第二个过程给两个参数 array 和 size 分别分配寄存器 x10 和 x11，并给 p 分配寄存器 x5。第二个过程的代码首先把指针 p 赋值为数组第一个元素的地址：

```
mv    x5, x10    // p = address of array[0]
```

以下代码是 for 循环体，将 0 存入 p：

```
loop2: sd    x0, 0(x5)    // Memory[p] = 0
```

这条指令实现循环体，因此以下代码迭代增加，即修改 p 以指向下一个双字：

```
addi    x5, x5, 8    // p = p + 8
```

在 C 语言中，指针自增 1 意味着将指针移动到下一个序列对象。由于 p 是声明为 long long int 的指向整数的指针，且每个整数占用 8 个字节，所以编译器将 p 增加 8。

接下来是循环测试。首先计算 array 的最后一个元素的地址。先将 size 乘以 8 得到它的字节地址：

```
slli    x6, x11, 3    // x6 = size * 8
```

然后将乘积加上数组的初始地址得到数组之后的第一个双字的地址：

```
add    x7, x10, x6      // x7 = address of array[size]
```

循环测试只需要判断 p 是否小于 array 的最后一个元素：

```
    bltu x5, x7, loop2     // if (p<&array[size]) go to loop2
```

完成所有片段后，可以给出将数组清零的指针版的代码：

```
            mv   x5, x10          // p = address of array[0]
loop2:      sd   x0, 0(x5)        // Memory[p] = 0
            addi x5, x5, 8        // p = p + 8
            slli x6, x11, 3       // x6 = size * 8
            add  x7, x10, x6      // x7 = address of array[size]
            bltu x5, x7, loop2    // if (p<&array[size]) go to loop2
```

与第一个示例一样，此代码假定 size 大于 0。

注意，该程序在循环的每次迭代中均计算数组末端地址，即使它没有改变。更快的代码版本是将该计算移出循环：

```
            mv   x5, x10          // p = address of array[0]
            slli x6, x11, 3       // x6 = size * 8
            add  x7, x10, x6      // x7 = address of array[size]
loop2:      sd   x0, 0(x5)        // Memory[p] = 0
            addi x5, x5, 8        // p = p + 8
            bltu x5, x7, loop2    // if (p < &array[size]) go to loop2
```

2.14.3　比较两个版本的 clear

并列比较两个代码序列以表明数组下标和指针之间的区别：

```
      li   x5, 0           // i = 0
loop1: slli x6, x5, 3      // x6 = i * 8
      add  x7, x10, x6     // x7 = address of array[i]
      sd   x0, 0(x7)       // array[i] = 0
      addi x5, x5, 1       // i = i + 1
      blt  x5, x11, loop1  // if (i < size) go to loop1
```

```
      mv   x5, x10         // p = address of array[0]
      slli x6, x11, 3      // x6 = size * 8
      add  x7, x10, x6     // x7 = address of array[size]
loop2: sd  x0, 0(x5)       // Memory[p] = 0
      addi x5, x5, 8       // p = p + 8
      bltu x5, x7, loop2   // if (p < &array[size]) go to loop2
```

左侧的版本在循环内必须具有“乘”和加，因为 i 增加了，每个地址必须由新的下标重新计算。右侧的内存指针版本直接增加指针 p。指针版本把实现缩放的移位操作和数组相关的加法操作移到循环外，从而将每次迭代执行的指令从 5 条减少到 3 条。这种手动优化对应于强度削弱（移位代替乘法）和循环变量消除（消除循环内的数组地址计算）的编译器优化。2.15 节描述了这两个优化及其他许多优化。

详细阐述　正如前面所提到的，C 编译器会增加一个检测来确保 size 大于 0。一种方法是在循环后用 `blt x0,x11,afterLoop` 跳转到该指令。

理解程序性能　我们曾经教育程序员要在 C 中使用指针来获得比数组更高的效率：“即使无法理解代码也要使用指针。”现代优化编译器可以为数组版本生成同样好的代码。如今大多数程序员更倾向于让编译器去做繁重的工作。

2.15　高级专题：编译 C 语言和解释 Java 语言

本节简要概述 C 编译器的工作原理及 Java 的执行方式。因为编译器将显著影响计算机性能，所以理解编译器技术是理解性能的关键。记住，“编译器的构建”课程通常需要 1 或 2 个学期的讲授，因此我们的介绍只涉及基本内容。

面向对 Java 这样的**面向对象语言**如何在 RISC-V 体系结构上执行感兴趣的读者。本节展示用于解释的 Java 字节码和前面章节中某些 C 程序段的 Java 版本的 RISC-V 代码，包括冒泡排序，并将涵盖 Java 虚拟机和即时（JIT）编译器。

面向对象语言：一种面向对象而非动作，或面向数据而非逻辑的编程语言。

本节剩余内容可在配套网站找到。

2.16 实例：MIPS 指令

与 RISC-V 最为相似的指令系统 MIPS 也起源于学术界，但现在已属于 Imagination Technologies。尽管 MIPS 先于 RISC-V 25 年（出现），但 MIPS 和 RISC-V 有着相同的设计理念。好在如果了解 RISC-V，那么拾起 MIPS 将非常容易。为了展现它们的相似性，图 2-29 比较了 RISC-V 和 MIPS 的指令格式。MIPS ISA 既有 32 位版本，又有 64 位版本，分别对应于 MIPS-32 和 MIPS-64。除了需要更大的地址外（64 位寄存器而非 32 位寄存器），这些指令系统几乎完全相同。以下是 RISC-V 和 MIPS 的共同特征：

- 对于两种体系结构，所有指令都是 32 位宽。
- 两者均有 32 个通用寄存器，其中一个寄存器硬连线为 0。
- 访问内存的唯一方法是通过两种体系结构的加载和存储指令。
- 与其他一些体系结构不同，在 MIPS 或 RISC-V 中，没有可以加载或存储许多寄存器的指令。
- 两者都具有寄存器等于零跳转和寄存器不等于零跳转的分支指令。
- 两个指令系统的寻址模式都适用于所有字长。

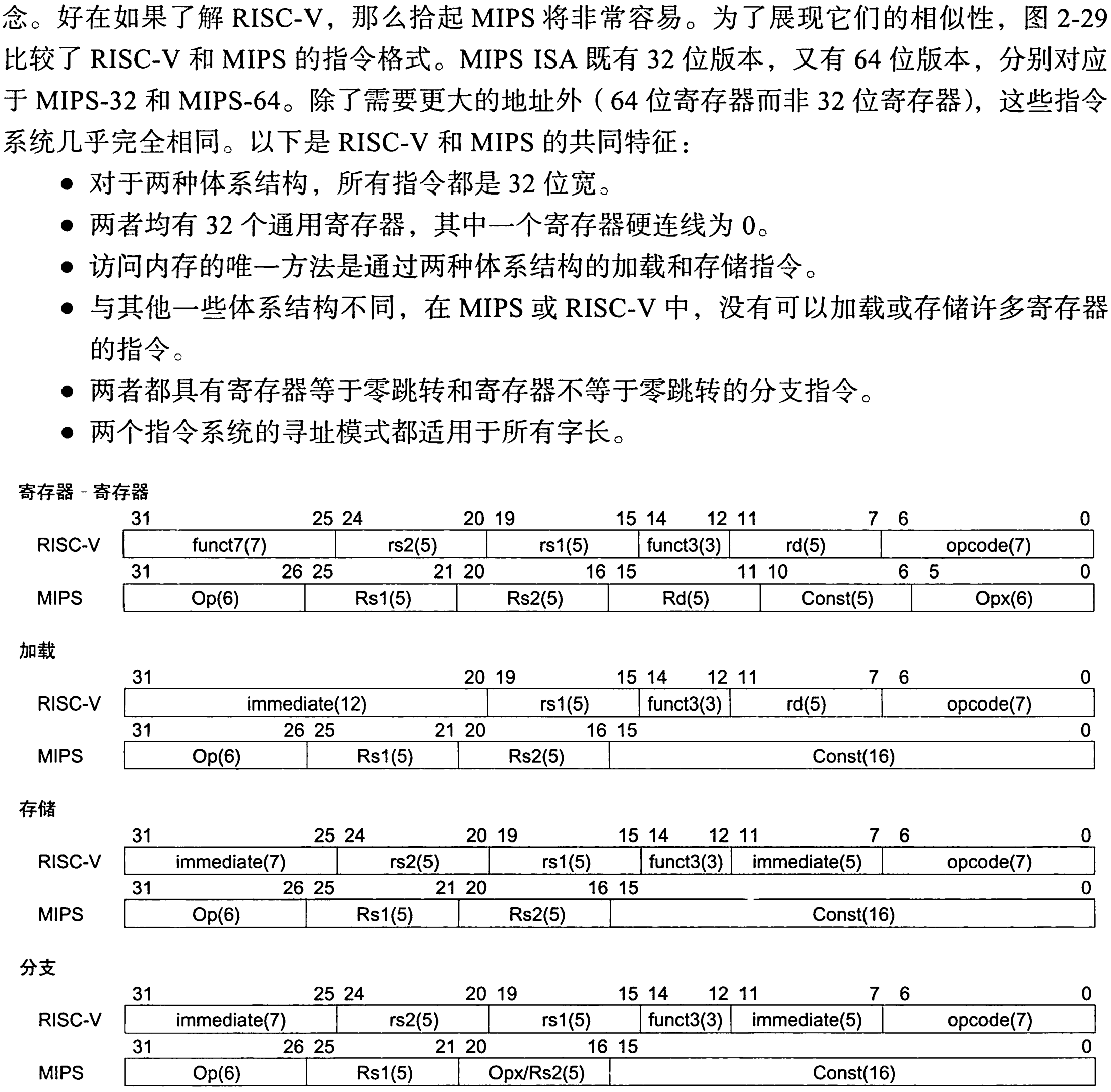


图 2-29 RISC-V 和 MIPS 的指令格式。相似在于两个指令集都有 32 个寄存器

RISC-V 和 MIPS 的主要区别之一是除相等或不等外的条件分支。RISC-V 仅提供分支指令来比较两个寄存器，而 MIPS 提供比较指令，该指令根据比较是否为真将寄存器设为 0 或 1。接着，程序员根据预期比较结果，在该比较指令后跟上一条等于或不等于零的分支指令。

遵循极简主义理念，MIPS 仅执行小于比较，让程序员改变操作数的顺序或改变分支测试条件以获得所有预期结果。MIPS 指令系统的小于比较指令存在有符号和无符号版本，分别为 `slt` 和 `sltu`。

除最常用的核心指令外，另一个主要区别是完整的 MIPS 指令系统要比 RISC-V 大得多，将在 2.18 节中看到。

2.17　实例：x86 指令

指令系统的设计者有时提供比 RISC-V 和 MIPS 更强大的操作。目标通常是减少程序执行的指令数。其风险在于：这种减少以简单为代价，因为指令执行更慢，所以会增加程序执行所需时间。这种缓慢可能是由于时钟周期更长或所需时钟周期数比简单序列更多。

情人眼里出西施。
Margaret Wolfe Hungerford, Molly Bawn, 1877

因此，通向复杂操作的道路困难重重。2.19 节展示了复杂性的陷阱。

2.17.1　Intel x86 的演变

RISC-V 和 MIPS 是单个团队一起工作推出的不同版本，这些体系结构的各个部分很好地配合在一起。x86 的情况却并非如此；它是几个独立团体开发的产品，他们在近 40 年的时间里不断改进该体系结构，向原始指令集添加新的功能，正如有人向打包好的袋子里添加衣服。以下是 x86 的重要里程碑。

通用寄存器：一种可以用于任何指令的地址或数据的寄存器。

- 1978 年：Intel 8086 体系结构宣布其是与当时已经成功的 8 位微处理器 Intel 8080 的汇编语言兼容的扩展。8086 是 16 位体系结构，所有内部寄存器都是 16 位宽。与 RISC-V 不同，它的寄存器都是专用的，因此不认为 8086 是**通用寄存器**（GPR）体系结构。
- 1980 年：Intel 8087 浮点协处理器发布。该体系结构在 8086 的基础上扩展了大约 60 条浮点指令。它用栈来替代寄存器（见 2.21 节和 3.7 节）。
- 1982 年：80286 扩展了 8086 体系结构，将地址空间增加到 24 位，创建了详细的内存映射和保护模型（见第 5 章），并添加了一些指令来完善指令集并控制保护模型。
- 1985 年：80386 将 80286 体系结构扩展到 32 位。除了具有 32 位寄存器和 32 位地址空间外，80386 还增加了新的寻址模式和附加操作。扩展的指令使 80386 几乎成为一个通用寄存器处理器。除分段寻址外，80386 还增加了对页的支持（见第 5 章）。与 80286 一样，80386 也具有无须修改即可执行 8086 程序的模式。
- 1989 ～ 1995 年：之后 1989 年的 80486、1992 年的 Pentium 和 1995 年的 Pentium Pro 旨在提高性能，只有四条指令添加到了用户可见的指令集中：三条有助于多处理技术（见第 6 章），以及一条条件传送指令。
- 1997 年：在 Pentium 和 Pentium Pro 发布后，Intel 宣布将用 MMX（多媒体扩展）扩展 Pentium 和 Pentium Pro 体系结构。这个 57 条指令的新指令集使用浮点栈来加速多媒体和通信应用程序。MMX 指令在传统的单指令多数据（Single Instruction, Multiple Data，SIMD）体系结构上一次处理多个短数据元素（见第 6 章）。Pentium Ⅱ没有引入任何新指令。
- 1999 年：Intel 添加了另外 70 条指令，标记 SSE（Streaming SIMD Extensions）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，将其宽度翻倍到 128 位，并添加一个单精度浮点数据类型。因此，可以并行执行四个 32 位浮点操作。为了提高内存性能，SSE 包括 cache 预取指令以及绕过 cache 并直接写入内存的流存储指令。

- 2001 年：Intel 又添加了另外 144 条指令，并标记为 SSE2。新数据类型是双精度算术，它允许并行执行成对的 64 位浮点操作。这 144 条指令几乎都是已存在的 MMX 和 SSE 指令的版本，它们并行运行 64 位数据。这种变化不仅可以实现更多的多媒体操作，而且相对于唯一的栈体系结构，它为编译器提供了不同的浮点操作目标。编译器可以选择将 8 个 SSE 寄存器用作浮点寄存器，如同其他计算机一样。这一变化大大提升了 Pentium 4 的浮点性能，Pentium 4 是第一款包含 SSE2 指令的微处理器。
- 2003 年：这次是非 Intel 的另一家公司改进了 x86 体系结构。AMD 宣布了一系列体系结构的扩展，将地址空间从 32 位增加到 64 位。类似于 1985 年在 80386 上从 16 位到 32 位地址空间的转换，AMD64 将所有寄存器扩展到 64 位。并将寄存器数增加到 16，将 128 位 SSE 寄存器的数增加到 16。ISA 的主要变化来自添加的一种称为长模式（long mode）的新模式，该模式用 64 位地址和数据重新定义了所有 x86 指令的执行。为了寻址更多的寄存器，它为指令添加了新的前缀。根据计算方式，长模式还添加了 4 到 10 条新指令，并去掉了 27 条旧指令。PC 相对寻址是另一个扩展。AMD64 仍然具有与 x86 相同的模式（遗产模式）以及将用户程序限制为 x86 但允许操作系统使用 AMD64 的模式（兼容模式）。与 HP / Intel IA-64 体系结构相比，这些模式能更好地过渡到 64 位寻址。
- 2004 年：Intel 认输并接受 AMD64，重新标记为 64 位扩展内存技术（Extended Memory 64 Technology，EM64T）。主要区别在于 Intel 添加了 128 位原子比较和交换指令，这个可能本应包含在 AMD64 中的指令。与此同时，Intel 宣布了另一代媒体扩展。SSE3 添加了 13 条指令，以支持复杂运算、结构数组上的图形操作、视频编码、浮点转换以及线程同步（见 2.11 节）。AMD 在后续芯片中添加了 SSE3，并向 AMD64 添加了缺少的原子交换指令，以维持与 Intel 的二进制兼容性。
- 2006 年：Intel 发布了 54 条新指令，作为 SSE4 指令集扩展的一部分。这些扩展执行的调整针对绝对差求和、数组结构的点积计算、窄数据到更宽数据的符号或零扩展、数目统计等。还增加了对虚拟机的支持（见第 5 章）。
- 2007 年：AMD 发布了 170 条指令，作为 SSE5 的一部分，包括 46 条基本指令集的指令增加了像 RISC-V 的 3 操作数指令。
- 2011 年：Intel 推出高级向量扩展，将 SSE 寄存器宽度从 128 位扩展到 256 位，从而重新定义了约 250 条指令并添加了 128 条新指令。

这段历史说明了兼容性这个“金手铐”对 x86 的影响，因为每一阶段存在的软件基础都至关重要，不会因重大体系结构变化而使其受到危害。

无论 x86 有多失败，该指令集在很大程度上驱动了计算机的 PC 时代，并且仍然支配着后 PC 时代的绝大部分。与 140 亿 ARM 芯片相比，每年制造 350M x86 芯片看似很少，但许多公司都想控制这个市场。无论如何，这个多变的家族带来了一种难以解释且无法喜爱的体系结构。

打起精神面对即将看到的内容！不必带着需要编写 x86 程序的担忧来阅读本节；相反，

本节的目的是让你熟悉世界上最流行的桌面体系结构的优缺点。

本节关心的是80386的32位子集，而不是整个16位、32位和64位指令集。我们将从寄存器和寻址模式开始说明，然后到整数操作，最后考虑指令编码。

2.17.2　x86寄存器和寻址模式

80386的寄存器展示了指令系统的进化（图2-30）。80386将所有16位寄存器（段寄存器除外）扩展为32位，给名称加上前缀E来表示32位版本。通常被称为GPR（General-Purpose Register，通用寄存器）。80386只包含8个GPR。这意味着RISC-V和MIPS程序可以使用四倍数量的寄存器。

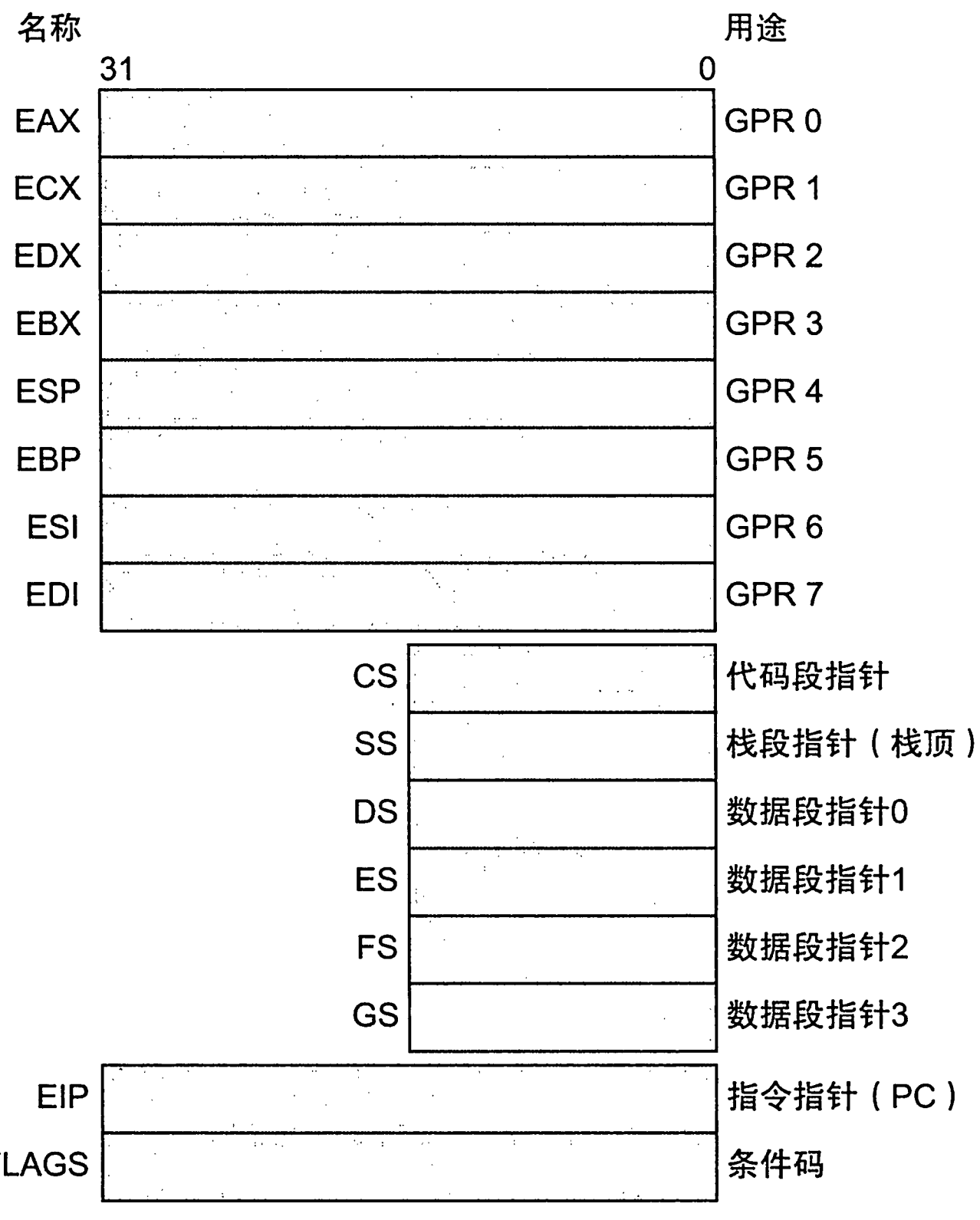


图2-30　80386寄存器组。从80386开始，前8个寄存器被扩展到32位，且可用作通用寄存器

图2-31展示了2操作数的算术、逻辑和数据传输指令。这里存在两个重要的区别。x86算术和逻辑指令中必须有一个操作数既作为源操作数，又作为目的操作数；RISC-V和MIPS允许源和目的操作数的寄存器分开。x86的这种限制对有限寄存器带来了更大的压力，因为必须修改一个源寄存器。第二个重要的区别在于其中一个操作数可以在存储器中。因此，与RISC-V和MIPS不同，实际上任何指令都可能有一个操作数在存储器中。

下面详细描述的数据存储器寻址模式在指令中提供两种位宽的地址。这些所谓的偏移（displacement）可以是8位或32位。

尽管存储器操作数可以使用任何寻址模式，但每种模式可以使用哪些寄存器存在限制。图2-32展示了x86寻址模式和各模式下哪些GPR不能使用，以及如何使用RISC-V指令得

到相同的效果。

源/目的操作数类型	第二个源操作数类型
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 2-31 x86 算术、逻辑和数据传输指令允许的操作数组合情况。唯一的限制是缺少存储器 - 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 2-30 中 14 个主要寄存器（除 EIP 和 EFLAGS）中的任意一个

模式	描述	寄存器限制	等价的RISC-V
寄存器间接寻址	地址在寄存器中	不能是ESP或EBP	`ld x10, 0(x11)`
8位或32位偏移量的基址寻址模式	地址是： 基址寄存器加上偏移量	不能是ESP	`ld x10, 40(x11)`

基址加比例下标寻址	地址是：
	基址+（$2^{比例}$×下标），
	其中比例为0、1、2或3
		基址：任意GPR
		下标：不能是ESP
			`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 0(x11)`

8位或32位偏移量的基址加比例下标寻址	地址是：
	基址+（$2^{比例}$×下标）+偏移量，
	其中比例为0、1、2或3
		基址：任意GPR
		下标：不能是ESP
			`slli x12, x12, 3`
			`add x11, x11, x12`
			`ld x10, 40(x11)`

图 2-32 受寄存器限制的 x86 32 位寻址模式及其等价的 RISC-V 代码。其中包含了在 RISC-V 或 MIPS 中所没有的基址加比例下标寻址模式，以避免乘以 8（比例因子 3）来将寄存器中的下标转换为字节地址（见图 2-26 和图 2-28）。比例因子 1 用于 16 位数据，比例因子 2 用于 32 位数据。比例因子 0 表示地址不缩放。如果在第二或第四种模式中偏移量长于 12 位，那么 RISC-V 等效模式将需要更多指令，通常用 `lui` 来载入偏移量的第 12 到 31 位，然后用 `add` 将其加到基址寄存器（Intel 为称为基址寻址的模式提供了两个不同的名称：基址和下标。但其本质相同，且在这里将其合并。）

2.17.3 x86 整数操作

8086 对 8 位（字节）和 16 位（字）数据类型提供支持。80386 在 x86 中增加了 32 位地址和数据（双字）。（AMD64 增加了 64 位地址和数据，称作四字；本节将关注 80386。）数据类型的区别也适用于寄存器操作以及存储器访问。

几乎所有操作都适用于 8 位数据和一个较长的数据大小。该大小由模式决定，为 16 位或 32 位。

显然，有些程序希望对所有三种大小的数据进行操作，因此 80386 体系结构提供了一种方便途径来指定每一种形式而不会显著扩展代码大小。它们认为 16 位或 32 位数据在大多数程序中占主导地位，因此设置一个默认的大尺寸是有意义的。这个默认数据大小由代码段寄存器中的一位设置。要重载默认数据大小，需在指令前附加一个 8 位前缀，以告诉机器该指令使用另一数据长度。

前缀解决方案是从 8086 借来的，8086 允许使用多个前缀来改变指令行为。最初的三个前缀忽略默认段寄存器，锁定总线以支持同步（见 2.11 节），或重复后续指令直到寄存器

ECX 减少到 0。最后一个前缀要配合一个字节传送指令以传送可变数目的字节。80386 还增加了一个前缀来改变默认地址长度。

x86 整数运算可分为四大类：

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、无条件分支、调用和返回。
4. 字符串指令，包括字符串传送和字符串比较。

除了算术和逻辑指令操作允许目的是寄存器或存储器位置外，前两个类别没有什么特别之处。图 2-33 展示了一些典型的 x86 指令及其功能。

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
movsl	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

图 2-33　一些典型的 x86 指令及其功能。常用操作列表如图 2-34 所示。CALL 将下一条指令的 EIP 保存在栈中（EIP 是 Intel 的 PC）

x86 上的条件分支基于条件代码或标志。条件代码被设置为操作的副作用，大多数用于将结果的值与 0 进行比较。然后，分支测试条件代码。PC 相对分支地址必须以字节数指定，因为与 RISC-V 和 MIPS 不同，80386 指令没有对齐限制。字符串指令是 x86 的 8080 系列的一部分，并且在大多数程序中通常不执行。它们通常比等效的软件程序慢（参见 2.19 节的谬误）。

图 2-34 列出了一些 x86 整数指令。大部分指令都同时具有字节和字格式。

指令	含义
控制	**条件和无条件分支**
jnz,jz	如果条件成立则跳转到EIP+8位偏移量；JNE（代替JNZ）和JE（代替JZ）两者之一
jmp	无条件跳转——8位或16位偏移量
call	子程序调用——16位偏移量；返回地址压栈
ret	从栈中弹出返回地址并跳转到该地址
loop	循环分支——自减ECX；如果ECX≠0，则跳转到EIP+8位偏移量处
数据传输	**在寄存器之间或寄存器与存储器之间移动数据**
move	在两个寄存器之间或寄存器与存储器之间移动数据
push,pop	源操作数压栈；从栈顶弹出操作数到寄存器
les	从存储器中载入ES和GPR中的一个

图 2-34　x86 中的典型操作。很多操作使用寄存器 – 存储器格式，其中一个源操作数或目的操作数可以是存储器，另一个可以是寄存器或立即数

指令	含义
算术，逻辑	**使用数据寄存器和存储器的算术与逻辑操作**
add,sub	将源操作数加到目的操作数；从目的操作数减去源操作数；寄存器-存储器格式
cmp	比较源操作数与目的操作数；寄存器-存储器格式
shl,shr,rcr	左移；逻辑右移；带条件码填充的循环右移
cbw	将EAX最右8位字节转换成EAX最右16位字
test	源操作数和目的操作数逻辑与，并设置标志位
inc,dec	目的操作数自增，目的操作数自减
or,xor	逻辑或；异或；寄存器-存储器格式
字符串	**字符串操作数间的移动；由重复前缀给定长度**
movs	通过递增ESI和EDI从源字符串复制到目的字符串；可以重复
lods	从字符串中取字节、字或双字到寄存器EAX

图 2-34 （续）

2.17.4 x86 指令编码

把最糟的放在最后，80386 中的指令编码很复杂，有很多不同的指令格式。当只有一个操作数时，80386 的指令可能从 1 个字节到 15 个字节。

图 2-35 展示了图 2-33 中几个示例指令的指令格式。操作码字节通常含有一位以表明操作数是 8 位还是 32 位。对于某些指令，操作码可能还包括寻址模式和寄存器；在许多具有“register = register op immediate”形式的指令中都是如此。其他指令使用“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），其中包含寻址模式信息。后置字节用于很多寻址存储器的指令。基址加比例下标寻址模式使用第二个后置字节，标记为“sc，index，base”（比例，下标，基址）。

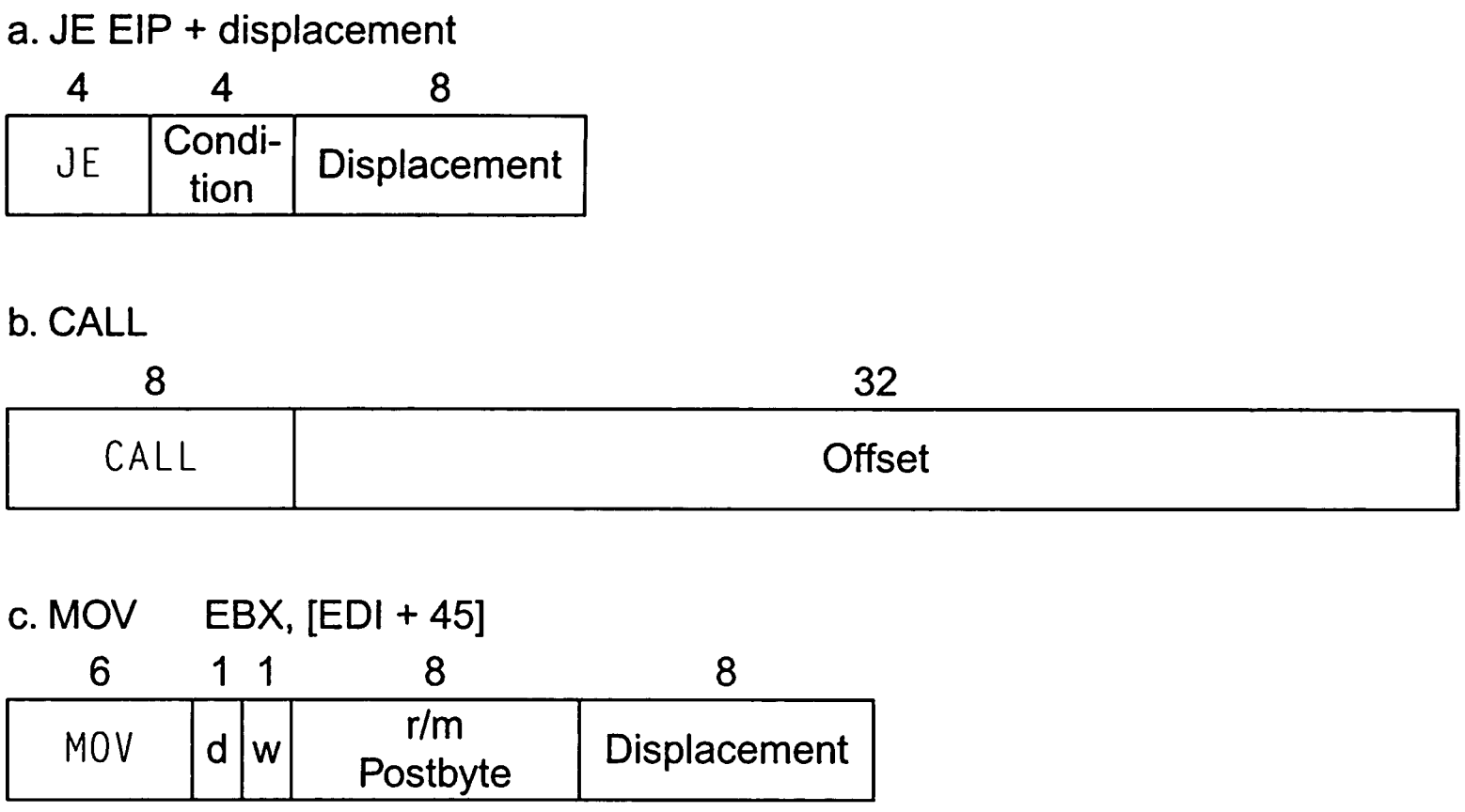
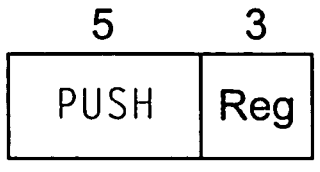
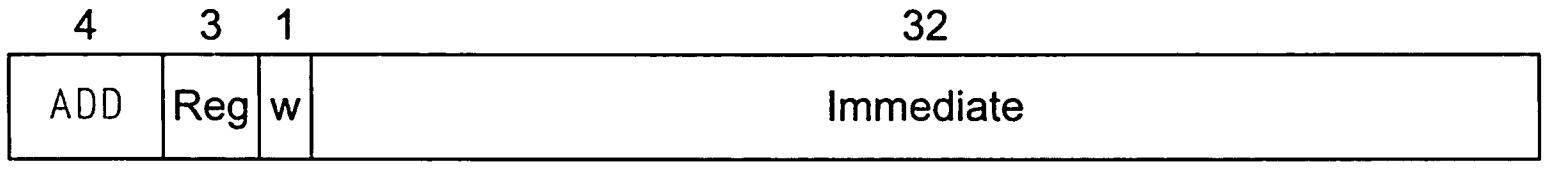


图 2-35 典型的 x86 指令格式。图 2-36 给出后置字节（postbyte）编码。很多指令包含 1 位的 w 字段，表明操作是字节还是双字。MOV 中的 d 字段可用于从存储器传出或传入到存储器，并表明传输方向。ADD 指令需要 32 位的立即数字段，因为在 32 位模式下，立即数为 8 位或 32 位。TEST 中的立即数字段长度为 32 位，因为在 32 位模式下没有 8 位的立即数检测。总的来说，指令长度可以从 1 到 15 个字节。较长的长度来自额外的 1 字节前缀，具有 4 字节立即数和 4 字节偏移地址，使用 2 字节的操作码，并使用比例下标寻址模式说明符，还需增加另一额外字节

d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

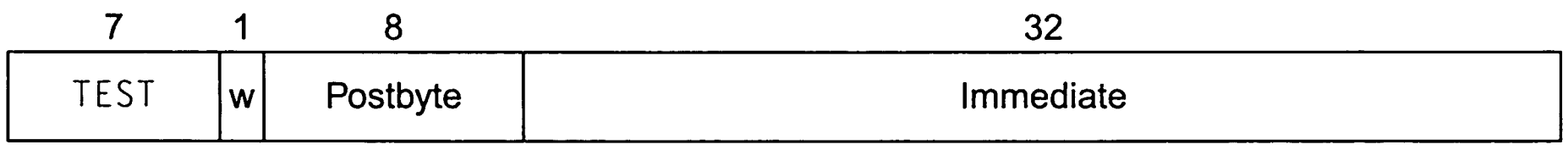


图 2-35 （续）

图 2-36 展示了 16 位和 32 位模式的两个后置字节地址说明符的编码。不幸的是，要完全理解哪些寄存器和哪些寻址模式可用，需要查看所有寻址模式的编码，有时甚至要看指令编码。

		16b	32b		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 2-36 x86 的第一个地址说明符编码：mod，reg，r/m。前 4 列表示 3 位 reg 字段的编码，该字段取决于操作码的 w 位以及机器处于 16 位模式（8086）还是 32 位模式（80386）。其余列解释了 mod 和 r/m 字段。3 位的 r/m 字段取决于 2 位 mod 字段的值和地址的大小。用于地址计算的寄存器列在第六和第七列中，在 mod = 0 时（mod = 1 时加上一个 8 位的偏移量，mod = 2 时加上一个 16 位或 32 位的偏移量）取决于寻址模式。例外情况有：①在 16 位模式下，当 mod = 1 或 mod = 2 时，r/m = 6 选择 BP 加上偏移量；②在 32 位模式下，当 mod = 1 或 mod = 2 时，r/m = 5 选择 EBP 加上偏移量；③在 32 位模式下，当 mod 不等于 3 时，r/m = 4，其中（sib）表示使用图 2-35 所示的比例下标寻址模式。当 mod = 3 时，r/m 字段指示一个寄存器，与 w 字段组合，使用与 reg 字段相同的编码

2.17.5 x86 总结

Intel 的 16 位微处理器先于其竞争对手的下一代体系结构两年（例如 Motorola 68000），这一先机致使 8086 被选作 IBM PC 的 CPU。Intel 工程师普遍承认 x86 比 RISC-V 和 MIPS 等计算机更难构建，但在 PC 时代，大型市场意味着 AMD 和 Intel 可以提供更多资源来帮助克服额外的复杂性。市场规模弥补了 x86 风格上的欠缺，从乐观的角度来看，这使其前景更好。

最常用的 x86 体系结构并不太难实现，自 1978 年以来，AMD 和 Intel 通过快速提高整点程序的性能证明了这一点。为了获得这样的性能，编译器必须避免使用高速结构，它们一

般难以实现。

然而在后 PC 时代，尽管已经具有相当多的体系结构和制造专业知识，x86 在个人移动设备中并不具备竞争力。

2.18 实例：RISC-V 指令系统的剩余部分

为了使指令系统体系结构广泛适用于各种计算机，RISC-V 架构师将指令系统划分为一个基本体系结构（base architecture）和几个扩展（extension）体系结构。每个都用字母表中的字母命名，基本体系结构命名为 I，表示整数。相对于当今其他流行指令系统，该基本体系结构仅有极少数指令，本章已经涵盖了这些相关指令。本节简要介绍基本体系结构以及五个标准扩展。

图 2-37 列出了 RISC-V 基本体系结构中的剩余指令。第一条指令为 auipc，用于 PC 相对存储器寻址。与 lui 指令一样，它保存一个 20 位的常数，对应于整数的第 12 位到第 31 位。auipc 用于将该数与 PC 相加并将结果写入寄存器。结合 addi 这样的指令，可以寻址 4 GiB 内的存储器的任意字节。这个特征对于*位置无关代码*（position-independent code）非常有用，无论从存储器的何处加载都可以正确执行。最常用于动态链接库。

RISC-V基本体系结构的特有指令

指令	名称	格式	描述
Add upper immediate to PC	auipc	U	立即数高20位与PC相加；将结果写到寄存器
Set if less than	slt	R	比较寄存器；将布尔结果写到寄存器
Set if less than, unsigned	sltu	R	比较寄存器；将布尔结果写到寄存器
Set if less than, immediate	slti	I	比较寄存器；将布尔结果写到寄存器
Set if less than immediate, unsigned	sltiu	I	比较寄存器；将布尔结果写到寄存器
Add word	addw	R	32位数字相加
Subtract word	subw	R	32位数字相减
Add word immediate	addiw	I	常数与32位数字相加
Shift left logical word	sllw	R	根据寄存器左移32位数字
Shift right logical word	srlw	R	根据寄存器右移32位数字
Shift right arithmetic word	sraw	R	根据寄存器算术右移32位数字
Shift left logical word immedate	slliw	I	根据立即数左移32位数字
Shift right logical word immediate	srliw	I	根据立即数右移32位数字
Shift right arithmetic word immediate	sraiw	I	根据立即数算术右移32位数字

图 2-37 在 RISC-V 指令系统中剩余的 14 条指令

接下来的四条指令比较两个整数，然后将比较后的布尔结果写入寄存器。slt 和 sltu 分别将两个寄存器作为有符号和无符号数进行比较，如果第一个值小于第二个值，则将 1 写到寄存器，否则写 0。slti 和 sltiu 执行相同的比较，但它们的第二个操作数为立即数。

剩余的指令应该看起来都很熟悉，因为它们的名称与本章讨论过的其他指令相同，但附加了字母 w，简称字（word）。这些指令与讨论过的相似命名指令执行相同的操作，只是这些指令仅使用其操作数的低 32 位，忽略第 32 到 63 位。另外，会产生符号扩展后的 32 位结果：即第 32 位到 63 位与第 31 位完全相同。RISC-V 架构师引入这些 w 指令是因为 32 位数的操作在 64 位地址的计算机上仍然很普遍。其主要原因是常用数据类型 int 在 Java 和大多数 C 语言的实现中仍是 32 位。

这就是基本体系结构！图 2-38 列出了五个标准扩展。第一个命名为 M，添加整数乘法和除法指令。第 3 章将介绍 M 扩展中的几条指令。

第二个扩展为 A，支持多处理器同步的原子存储器操作。2.11 节中介绍的保留加载（lr.d）和条件存储（sc.d）指令都属于 A 扩展。还包括 32 位字（lr.w 和 sc.w）版本。剩余 18 条指令是常见同步模式的优化，如原子交换和原子添加，但没有在保留加载和条件存储上添加任何其他功能。

RISC-V基本体系结构与扩展

助记符	描述	指令数
I	基本体系结构	51
M	整数乘法/除法	13
A	原子操作	22
F	单精度浮点	30
D	双精度浮点	32
C	压缩指令	36

图 2-38　RISC-V 指令系统体系结构分为基本 ISA（称作 I）以及五个标准扩展：M、A、F、D 和 C

第三和第四个扩展 F 和 D 提供浮点数操作，将在第 3 章中讲述。

最后一个扩展 C 不提供任何新功能。相反，它采用最常见的 RISC-V 指令，如 addi，并提供长度仅为 16 位而非 32 位的等效指令。从而允许程序用更少的字节表示，这可以降低成本（将在第 5 章中看到），还能提升性能。为了适应 16 位，新指令对其操作数有限制：例如，某些指令只能访问 32 个寄存器中的一部分，且立即数字段更窄。

总而言之，RISC-V 的基本体系结构及其扩展共有 184 条指令以及 13 条系统指令（将在第 5 章末尾介绍）。

2.19　谬误与陷阱

谬误：更强大的指令意味着更高的性能。

Intel x86 的一个强大之处在于前缀，它可以改变后续指令的执行。前缀可以重复后续指令，直到计数器递减至 0。因此，在存储器中移动数据，似乎自然的指令序列就是使用带有重复前缀的 move 来执行 32 位存储器到存储器的传送。

另一种方法是使用所有计算机中都有的标准指令，将数据加载到寄存器中，然后将寄存器存回存储器。这种方法的第二版是复制代码以减少循环开销，复制约快 1.5 倍。第三版使用更大的浮点寄存器而不是 x86 的整数寄存器，复制要比复杂的传送指令快 2.0 倍。

谬误：用汇编语言编程以获得最高性能。

曾经有一段时间，编程语言的编译器常常产生低级的指令序列；编译器日渐复杂意味着编译产生的代码和手工生成的代码之间的差距正在快速缩小。事实上，为了与当今的编译器竞争，汇编语言程序员需要彻底理解第 4 章和第 5 章中的概念（处理器流水线和存储器层次结构）。

编译器和汇编程序员之间的斗争正在逐渐消失。例如，C 为程序员提供了一个机会来提示编译器哪些变量要保留在寄存器中，哪些变量要换出到存储器中。当编译器不能很好地分配寄存器时，这些提示对性能来说至关重要。实际上，一些旧的 C 课本花费大量的时间给出了有效使用寄存器提示的例子。今天的 C 编译器通常忽略这些提示，因为编译器在分配方面比程序员做得更好。

即使手工编写能产生更快的代码，使用汇编语言编写的危险在于编码和调试所花费的时间、可移植性差以及难以维护。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越长，编码需要的时间就越长，显然使用汇编语言编写程序比使用 C 或 Java 编写的程序要长