2.7.3 case/switch 语句

大多数编程语言都包含 case 或 switch 语句，允许程序员根据某个值选择多个分支中的一个。实现 switch 的最简单方法是通过一系列的条件测试，将 switch 语句转换成 if-then-else 语句。

有时，另一种更有效的方法是编码形成指令序列的地址表，称为**分支地址表**或**分支表**，程序只需要索引到表中，然后跳转到合适的指令序列。因此，分支表只是一个双字数组，其中包含与代码中的标签对应的地址。该程序将分支表中的相应条目加载到寄存器中，然后需要使用寄存器中的地址进行跳转。为了支持这种情况，RISC-V 这类指令系统包含一个间接跳转指令，该指令对寄存器中指定的地址执行无条件跳转。在 RISC-V 中，跳转－链接指令（jalr）用于此目的。我们将在下一节中看到这种多功能指令更多常见的使用方式。

分支地址表：也称作分支表，一种包含了不同指令序列地址的表。

硬件/软件接口　虽然在类似 C 和 Java 这样的编程语言中有很多决策和循环的语句，但在指令系统级别实现它们的基础语句是条件分支。

自我检测

I. C 语言中有许多决策和循环语句，但在 RISC-V 中却很少。下面的各项有没有阐明这种差别？为什么？

1. 更多的决策语句让代码更易于阅读和理解。
2. 更少的决策语句简化了负责执行的底层任务。
3. 更多的决策语句意味着的更少的代码量，这缩减了编程的时间。
4. 更多的决策语句意味着更少的代码量，这意味着执行更少的操作。

II. 为什么 C 语言提供了两种 AND 操作（& 和 &&）和两种 OR 操作（| 和 ||），而 RISC-V 没有？

1. 逻辑操作 AND 和 OR 对应于 & 和 |，而条件分支对应于 && 和 ||。
2. 上面说反了，&& 和 || 对应于逻辑操作，而 & 和 | 对应于条件分支。
3. 它们是多余的，表示同样的意思：&& 和 || 都是简单地继承于 C 语言的前身——B 语言。

2.8 计算机硬件对过程的支持

过程（procedure）或函数是编程人员用于结构化编程的一种工具，两者均有助于提高程序的可理解性和代码的可重用性。过程允许程序员一次只专注于任务的一部分；参数可以传递数值并返回结果，因此用以充当过程和其余程序与数据之间的接口。2.15 节描述了 Java 中过程的等效表示，但 Java 对计算机的要求和 C 完全相同。过程是用软件实现抽象的一种方式。

过程：一个根据给定参数执行特定任务的已存储的子程序。

跳转－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在寄存器（在 RISC-V 中通常是 x1）中的指令。

可以把过程想象成一个携带秘密计划离开的侦探，他获取资源，执行任务，掩盖踪迹，然后带着预期结果返回原点。一旦任务完成，则再无任何干扰。更重要的是，侦探只在“需要知道”的基础上运作，因此侦探不能对雇主有任何臆断。

同样，在执行过程时，程序必须遵循以下六个步骤：

1. 将参数放在过程可以访问到的位置。

2. 将控制转交给过程。

3. 获取过程所需的存储资源。

4. 执行所需的任务。

5. 将结果值放在调用程序可以访问到的位置。

6. 将控制返回到初始点，因为过程可以从程序中的多个点调用。

如上所述，寄存器是计算机中访问数据最快的存储位置，因此期望尽可能多地使用它们。RISC-V 软件为过程调用分配寄存器时遵循以下约定：

- x10 ~ x17：八个参数寄存器，用于传递参数或返回值。
- x1：一个返回地址寄存器，用于返回到起始点。

除了分配这些寄存器之外，RISC-V 汇编语言还包含一个仅用于过程的指令：跳转到某个地址的同时将下一条指令的地址保存到目标寄存器 rd。**跳转 – 链接指令**（jal）写作：

```
jal x1, ProcedureAddress        // jump to
   ProcedureAddress and write return address to x1
```

指令中的链接部分表示指向调用点的地址或链接，以允许该过程返回到合适的地址。存储在寄存器 x1 中的这个“链接”被称为**返回地址**。返回地址是必需的，因为同一过程可能在程序的不同部分被调用。

返回地址：指向调用点的链接，允许过程返回到合适的地址；在 RISC-V 中它被存储在寄存器 x1 中。

为了支持这种情况下的过程返回，类似 RISC-V 的计算机使用了间接跳转（如上述跳转 – 链接指令（jalr）），用以处理 case 语句：

```
jalr x0,  0(x1)
```

调用者：启动过程并提供必要参数值的程序。

正如所期望的那样，寄存器跳转 – 链接指令跳转到存储在寄存器 x1 中的地址。因此，调用程序或称为**调用者**将参数值放入 x10 ~ x17 中，并使用 jal x1，X 跳转到过程 X（有时称为**被调用者**）。被调用者执行计算，将结果放在相同的参数寄存器中，并使用 jalr x0，0(x1) 将控制返还给调用者。

被调用者：根据调用者提供的参数执行一系列已存储的指令的过程，然后将控制权返还给调用者。

在存储程序概念中，需要一个寄存器来保存当前执行指令的地址。由于历史原因，这个寄存器总是被称为**程序计数器**（program counter）（在 RISC-V 体系结构中缩写为 PC），尽管其更合理的名称可能是指令地址寄存器。jal 指令实际上将 PC + 4 保存在其指定寄存器（通常为 x1）中，以链接到后续指令的字节地址来设置过程返回。

程序计数器：包含程序中正在执行指令地址的寄存器。

详细阐述 通过使用 x0 作为目标寄存器，跳转 – 链接指令也可用于实现过程内的无条件跳转。由于 x0 硬连线为零，其效果是丢弃返回地址：

```
jal x0, Label  // unconditionally branch to Label
```

2.8.1 使用更多的寄存器

假设对于一个过程，编译器需要比 8 个参数寄存器更多的寄存器。由于在任务完成后必须掩盖踪迹，调用者所需的所有寄存器都必须恢复到调用该过程之前所存储的值。这种情况是需要将寄存器换出到存储器中的一个例子，正如 2.3.1 节的“硬件 / 软件接口”部分所述。

栈：一种被组织成后进先出队列并用于寄存器换出的数据结构。

换出寄存器的理想数据结构是**栈**（stack）——一种后进先出的队列。栈需要一个指向栈中最新分配地址的指针，以指示下一个过程应该放置换出寄存器的位置或寄存器旧值的存放位置。在 RISC-V 中，**栈指针**（stack pointer）是寄存器 x2，也称为 sp。栈指针按照每个被保存或恢复的寄存器按双字进行调整。栈应用非常广泛，因而传送数据到栈或从栈传输数据都具有专业术语：将数据放入栈中称为**压栈**，从栈中移除数据称为**弹栈**。

栈指针：指示栈中最新分配的地址的值，用于指示应该被换出的寄存器的位置，或寄存器旧值的存放位置。在 RISC-V 中为寄存器 sp 或 x2。

压栈：向栈中添加元素。

弹栈：从栈中移除元素。

按照历史惯例，栈按照从高到低的地址顺序“增长”。这就意味着可以通过减栈指针将值压栈；通过增加栈指针缩小栈，从而弹出栈中的值。

例题｜编译一个没有调用其他过程的 C 过程

将 2.2 节的例题转化为一个 C 过程：

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

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

编译后的 RISC-V 汇编代码是什么呢？

答案｜参数变量 g、h、i 和 j 对应于参数寄存器 x10、x11、x12 和 x13，f 对应于 x20。编译后的程序从如下过程标号开始：

```
leaf_example:
```

下一步是保存该过程使用的寄存器。过程体中的 C 赋值语句与之前的例题相同，使用两个临时寄存器（x5 和 x6）。因此，需要保存三个寄存器：x5、x6 和 x20。通过在栈中创建三个双字（24 字节）空间并将数据存入，实现将旧值“压”入栈中：

```
addi sp, sp, -24          // adjust stack to make room for 3 items
sd   x5, 16(sp)           // save register x5 for use afterwards
sd   x6, 8(sp)            // save register x6 for use afterwards
sd   x20, 0(sp)           // save register x20 for use afterwards
```

图 2-10 展示了过程调用之前、之中和之后栈的情况。

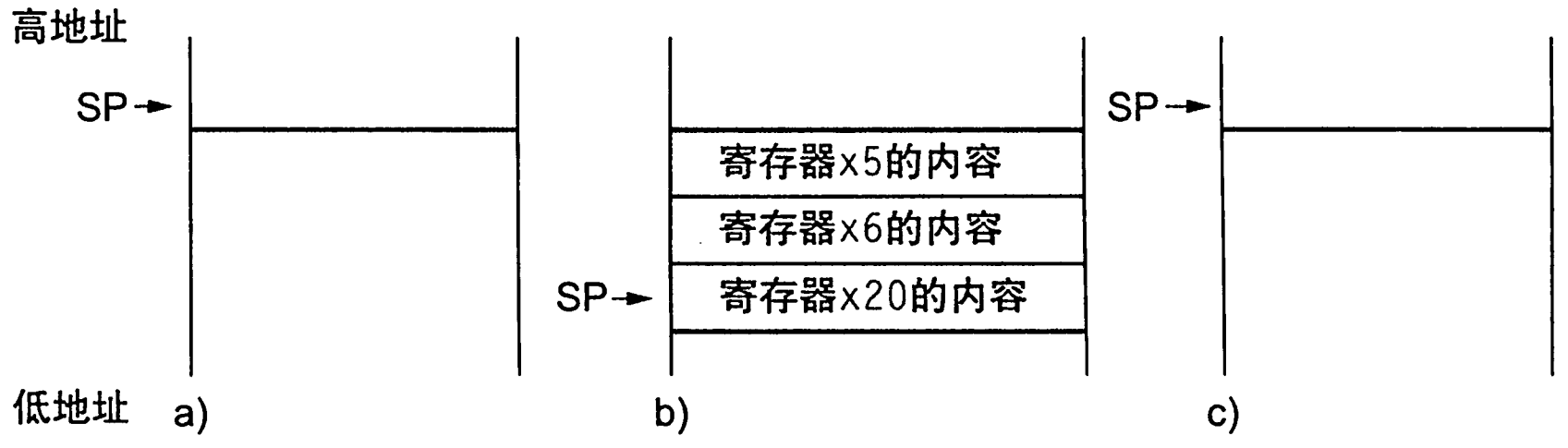

图 2-10　过程调用之前（a）、之中（b）和之后（c）栈指针以及栈的值。栈指针总是指向栈“顶”，或者图中栈的最后一个双字

以下三条语句对应于 2.2 节例题后的过程体：

```
add x5, x10, x11    // register x5 contains g + h
add x6, x12, x13    // register x6 contains i + j
sub x20, x5, x6     // f = x5 - x6, which is (g + h) - (i + j)
```

为了返回 f 的值，将其复制到一个参数寄存器中：

```
addi x10, x20, 0 // returns f (x10 = x20 + 0)
```

在返回之前，通过从栈中“弹出”数据来恢复寄存器的三个旧值：

```
ld x20, 0(sp)     // restore register x20 for caller
ld x6, 8(sp)      // restore register x6 for caller
ld x5, 16(sp)     // restore register x5 for caller
addi sp, sp, 24  // adjust stack to delete 3 items
```

通过一个使用返回地址的跳转寄存器结束过程：

```
jalr x0, 0(x1)        // branch back to calling routine
```

先前示例使用了临时寄存器，并假设其旧值必须被保存和恢复。为了避免保存和恢复一个其值从未被使用过的寄存器（通常为临时寄存器），RISC-V 软件将 19 个寄存器分成两组：

- x5 ~ x7 以及 x28 ~ x31：临时寄存器，在过程调用中不被被调用者（被调用的过程）保存。
- x8 ~ x9 以及 x18 ~ x27：保存寄存器（saved register），在过程调用中必须被保存。（一旦使用，由被调用者保存并恢复）

这一简单约定减少了寄存器换出。在上述例子中，由于调用者不希望在过程调用中保存寄存器 x5 和 x6，可以从代码中去掉两次存储和两次载入。但仍须保存并恢复 x20，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子（leaf）过程。如果所有过程都是叶子过程，情况将会变得简单，但事实并非如此。正如一个侦探任务的一部分可能是雇佣其他侦探一样，被雇佣的侦探进而雇佣更多的侦探，过程调用其他过程也是如此。更进一步，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时需要小心一样，在调用非叶子过程时必须更加注意。

例如，假设主程序调用过程 A，参数为 3，将值 3 存入寄存器 x10 然后使用 jal x1, A。再假设过程 A 通过 jal x1, B 调用过程 B，参数为 7，也存入 x10。由于 A 尚未结束任务，所以寄存器 x10 的使用存在冲突。同样在寄存器 x1 中的返回地址也存在冲突，因为它现在具有 B 的返回地址。除非采取措施阻止这类问题发生，否则该冲突将导致过程 A 无法返回其调用者。

一种解决方法是将其他所有必须保存的寄存器压栈，就像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（x10 ~ x17）或临时寄存器（x5 ~ x7 和 x28 ~ x31）压栈。被调用者将返回地址寄存器 x1 和被调用者使用的保存寄存器（x8 ~ x9 和 x18 ~ x27）压栈。调整栈指针 sp 以计算压栈寄存器的数量。返回时，从存储器中恢复寄存器并重新调整栈指针。

例题 | 编译一个递归 C 过程，演示嵌套过程的链接

处理一个计算阶乘的递归过程：

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

RISC-V 汇编代码是什么呢?

答案 | 参数变量 n 对应参数寄存器 x10。编译后的程序从过程的标签开始，然后在栈中保存两个寄存器，返回地址和 x10：

```
fact:
  addi sp, sp, -16   // adjust stack for 2 items
    sd x1, 8(sp)     // save the return address
    sd x10, 0(sp)    // save the argument n
```

第一次调用 fact 时，sd 保存程序中调用 fact 的地址。下面两条指令测试 n 是否小于 1，如果 n ≥ 1 则跳转到 L1。

```
addi    x5, x10, -1        // x5 = n - 1
bge     x5, x0, L1         // if (n - 1) >= 0, go to L1
```

如果 n 小于 1，fact 将 1 放入一个值寄存器中以返回 1：它将 1 加 0 并将和存入 x10 中。然后从栈中弹出两个已保存的值并跳转到返回地址：

```
addi    x10, x0, 1   // return 1
addi    sp, sp, 16   // pop 2 items off stack
jalr    x0, 0(x1)    // return to caller
```

在从栈中弹出两项之前，可以加载 x1 和 x10。因为当 n 小于 1 时 x1 和 x10 不会改变，所以跳过这些指令。

如果 n 不小于 1，则参数 n 递减，然后用递减后的值再次调用 fact：

```
L1: addi x10, x10, -1  // n >= 1: argument gets (n - 1)
    jal  x1, fact      // call fact with (n - 1)
```

下一条指令是 fact 的返回位置，其结果在 x10 中。现在旧的返回地址和旧的参数与栈指针一起被恢复：

```
addi  x6, x10, 0    // return from jal: move result of fact
                       (n - 1) to x6:
ld    x10, 0(sp)    // restore argument n
ld    x1, 8(sp)     // restore the return address
addi  sp, sp, 16    // adjust stack pointer to pop 2 items
```

接下来，参数寄存器 x10 得到旧参数与 fact(n-1) 结果的乘积，目前在 x6 中。假设有一个乘法指令可用，尽管在第 3 章才会涉及：

```
mul     x10, x10, x6      // return n * fact (n - 1)
```

最后，fact 再次跳转到返回地址：

```
jalr     x0, 0(x1)      // return to the caller
```

硬件 / 软件接口 C 变量通常指一个存储位置，其解释取决于其类型（type）和存储方式（storage class）。示例类型包括整型和字符型（见 2.9 节）。C 语言有两种存储方式：动态的（automatic）和静态的（static）。动态变量位于过程中，并在程序退出时失效。静态变量从

过程进入到退出始终存在。在所有过程之外声明的 C 变量以及使用关键字 static 声明的所有变量，都被认为是静态的。其余的都是动态的。为了简化静态数据的访问，一些 RISC-V 编译器保留一个寄存器 x3 用作全局指针（global pointer）或 gp。

全局指针：指向静态数据区的保存寄存器。

图 2-11 总结了过程调用中的保存对象。需要注意的是，有些方案也保存了栈，以确保调用者在弹栈时取回与压栈时相同的数据。sp 以上的栈通过确保被调用者不在其上进行写入来保存；sp 本身就是由被调用者将其被减去的值重新加上来保存的；并且其他寄存器通过将它们保存到栈（若被使用）并从栈中将其恢复来进行保存。

保存	不保存
保存寄存器：x8～x9，x18～x27	临时寄存器：x5～x7，x28～x31
栈指针寄存器：x2(sp)	参数/结果寄存器：x10～x17
帧指针：x8(fp)	
返回地址：x1(ra)	
栈指针以上的栈	栈指针以下的栈

图 2-11 过程调用中保存的及不保存的对象。如果软件依赖于全局指针寄存器（将在下面讨论），则也需要保存

2.8.3 在栈中为新数据分配空间

最后一点复杂性在于栈也用于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段称为**过程帧**或**活动记录**。图 2-12 展示了过程调用之前，期间和之后栈的状态。

过程帧：也称作活动记录。栈中包含过程保存的寄存器和局部变量的段。

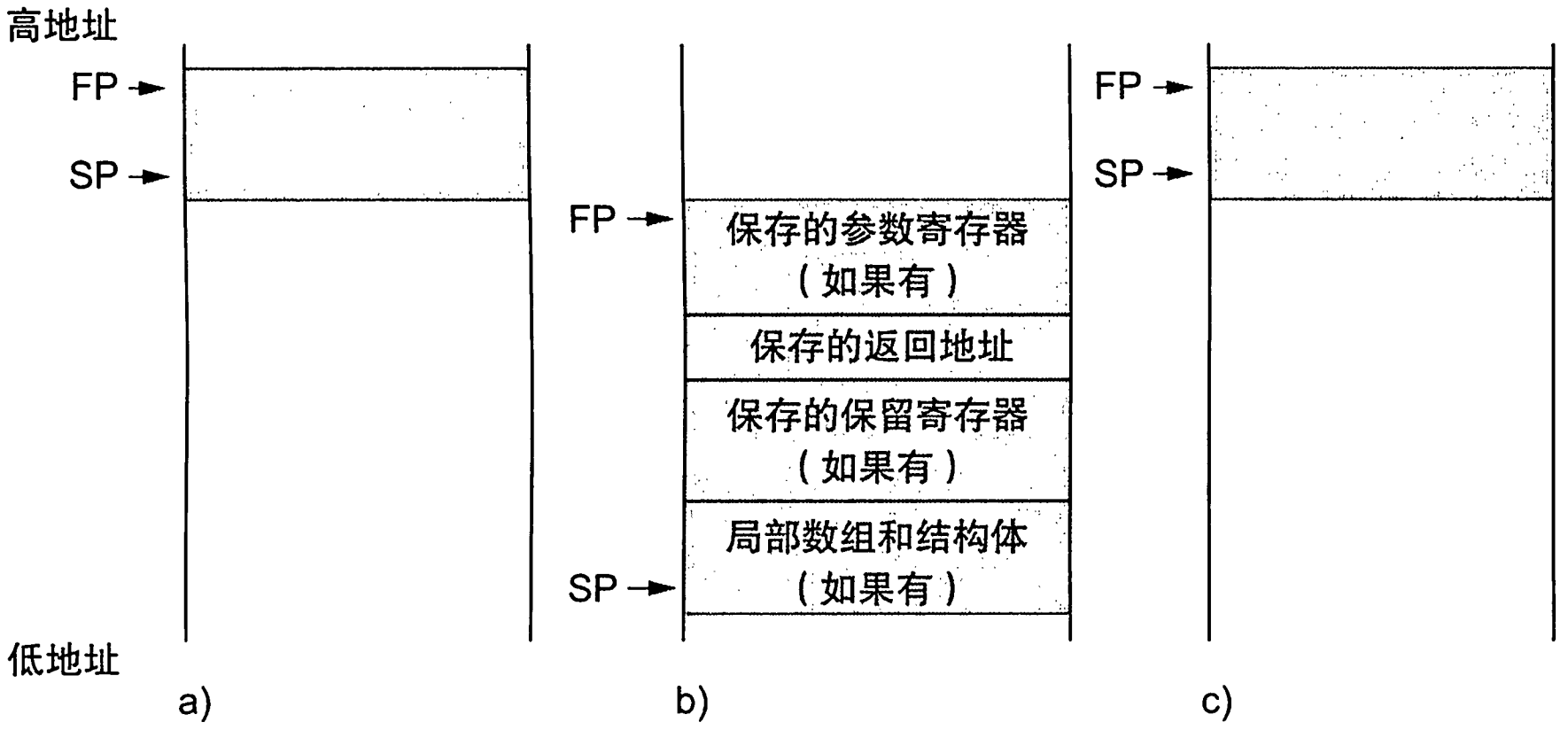

图 2-12 过程调用之前（a）、之中（b）和之后（c）栈的分配情况。帧指针（fp 或 x8）指向帧的第一个双字，通常是保存的参数寄存器，栈指针（sp）指向栈顶。调整栈以容纳所有保存的寄存器和常驻存储器的局部变量。由于栈指针在程序执行过程中可能会发生改变，对程序员而言，尽管只需要使用栈指针和少量的地址运算即可完成对变量的引用，但通过稳定的帧指针可以更容易地引用变量。如果在过程中栈内没有局部变量，编译器将不设置和不恢复帧指针以节省时间。当使用帧指针时，在调用中使用 sp 的地址进行初始化，且可以使用 fp 恢复 sp。相关内容也可以在本书 RISC-V 参考数据卡的第 4 列中找到

一些 RISC-V 编译器使用**帧指针** fp 或者寄存器 x8 来指向过程帧的第一个双字。栈指针在过程中可能会发生改变，因此对存储器中局部变量的引用可能会有不同的偏移量，具体取决于它们在过程中的位置，从而使过程更难理解。帧指针在过程中为局部变量引用提供一个稳定的基址寄存器。注意，不管是否使用显式的帧指针，栈上都会显示一条活动记录。我们可以通过维护稳定的 sp 来减少对 fp 的使用：在示例中，仅在进入和退出过程时才调整栈。

帧指针：指向给定过程的局部变量和保存的寄存器地址的值。

2.8.4　在堆中为新数据分配空间

除了动态变量（对于过程局部有效）之外，C 程序员还需要为静态变量和动态数据结构分配内存空间。图 2-13 展示了运行 Linux 操作系统时 RISC-V 分配内存的约定。栈从用户地址空间的高端开始（见第 5 章）并向下扩展。低端内存的第一部分是保留的，之后是 RISC-V 机器代码，通常称为**代码段**（text segment）。在此之上是静态数据段（static data segment），用于存放常量和其他静态变量。虽然数组具有固定长度，且因此可与静态数据段很好地匹配，但像链表等数据结构往往会随生命周期增长和缩短。存放这类数据结构（数组和链表）的段通常称为堆（heap），它放在内存中。注意，这种分配允许栈和堆相向而长，从而随着这两个段的此消彼长达到内存的高效使用。

代码段：UNIX 目标文件的段，包含源文件中例程的机器语言代码。

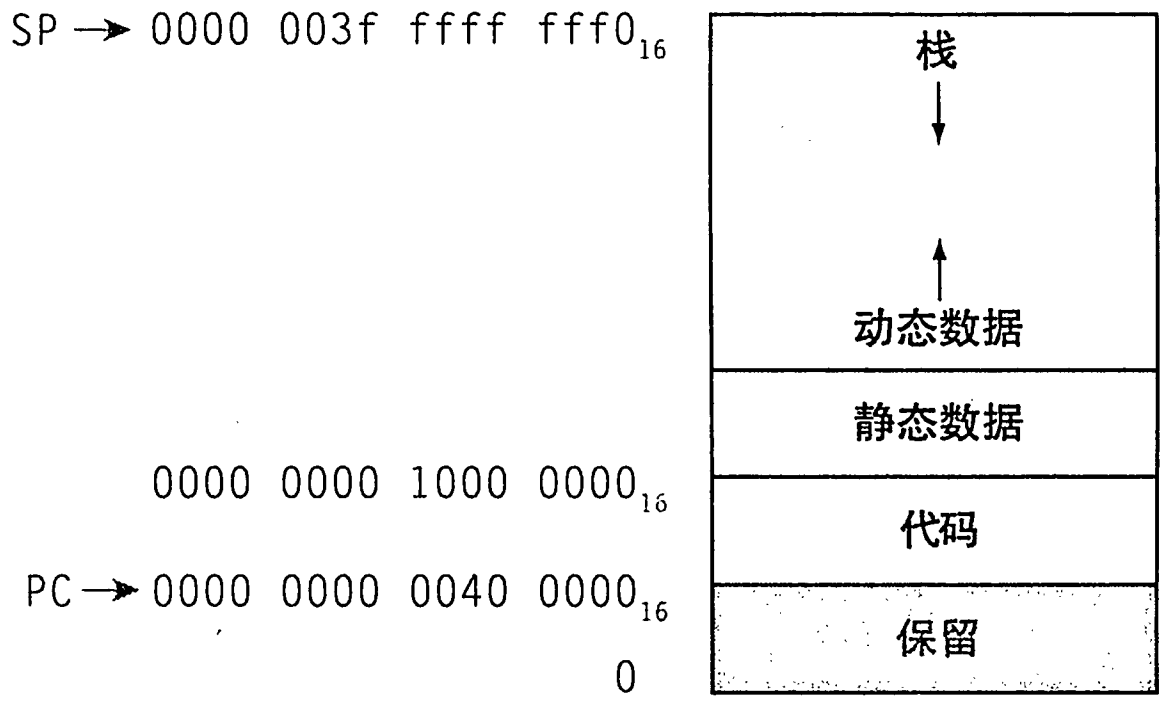

图 2-13　程序和数据的 RISC-V 内存分配。这些地址只是一种软件规定，而不是 RISC-V 体系结构的一部分。当给定 64 位体系结构时，用户地址空间设置为可能的 2^{64} 总地址空间的 2^{38}（见第 5 章）。栈指针初始化为 0000 003f ffff fff0$_{16}$ 并向下增长至数据段。在另一端，程序代码（图中的“代码”）从 0000 0000 0040 0000$_{16}$ 开始。静态数据始于文本段末；在本例中，假设地址是 0000 0000 1000 0000$_{16}$。之后是动态数据，由 C 中的 malloc 和 Java 中的 new 分配。它在称为堆的区域中向栈的方向增长。关于这点可参考本书 RISC-V 参考数据卡的第 4 列

C 语言通过显式函数调用来分配和释放堆上的空间。malloc( ) 在堆上分配空间并返回指向它的指针，free( ) 释放指针所指向的堆空间。C 程序控制内存分配，这是许多常见和困难 bug 的根源。忘记释放空间会导致“内存泄漏”，最终耗尽大量内存，可能导致操作系统崩溃。过早释放空间会导致“悬空指针”，这可能导致指针指向程序未曾打算访问的位置。Java 使用自动内存分配和垃圾回收机制来避免这类错误。

图 2-14 总结了 RISC-V 汇编语言的寄存器约定。这个约定是加速经常性事件的另一个例子：大多数过程可以使用多达 8 个参数寄存器、12 个保留寄存器和 7 个临时寄存器而无须进入内存。

名称	寄存器号	用法	调用时是否保存
x0	0	常数0	不适用
x1（ra）	1	返回赋值（链接寄存器）	是
x2（sp）	2	栈指针	是
x3（gp）	3	全局指针	是
x4（tp）	4	线程指针	是
x5～x7	5～7	临时	否
x8～x9	8～9	保存	是
x10～x17	10～17	参数/结果	否
x18～x27	18～27	保存	是
x28～x31	28～31	临时	否

图 2-14 RISC-V 寄存器约定。此信息也可见本书 RISC-V 参考数据卡的第 2 列

详细阐述 如果参数超过 8 个怎么办？RISC-V 约定将栈中额外的参数放在帧指针的上方。过程期望前 8 个参数在寄存器 x10 到 x17 中，其余参数在内存中，可通过帧指针寻址。

如图 2-12 的标题所述，帧指针的方便性在于对过程中栈内变量的所有引用都具有相同的偏移。但是，帧指针并不是必需的。RISC-V C 编译器仅在改变了栈指针的过程中使用帧指针。

详细阐述 一些递归过程可以不使用递归而用迭代实现。迭代可以通过消除与递归调用相关的开销来显著提高性能。例如，考虑一个用于求和的过程：

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

考虑过程调用 sum(3,0)。这将导致对 sum(2,3)、sum(1,5) 和 sum(0,6) 的递归调用，然后结果 6 将返回四次。这种求和的递归调用称为尾调用（tail call），而这个使用尾递归的示例可以用迭代高效实现（假设 x10=n，x11=acc，结果放入 x12）：

```
sum: ble x10, x0, sum_exit   // go to sum_exit if n <= 0
     add x11, x11, x10       // add n to acc
     addi x10, x10, -1       // subtract 1 from n
     jal x0, sum             // jump to sum
sum_exit:
     addi x12, x11, 0        // return value acc
     jalr x0, 0(x1)          // return to caller
```

自我检测 以下关于 C 和 Java 的描述哪个通常是正确的？

1. C 程序员显式管理数据，而 Java 则是自动管理。
2. C 会比 Java 导致更多的指针错误和内存泄漏错误。

2.9　人机交互

计算机的发明是为了数字计算，但很快被用于商业方面的文本处理。当前大多数计算机使用字节来表示字符，也就是每个人都遵循的表示方法 ASCII（American Standard Code for Information Interchange）。图 2-15 总结了 ASCII 码。

!(@ | = > (wow open tab at bar is great)
键盘诗“Hatless Atlas”的第 4 行，1991（一些对 ASCII 字符的命名：“!”是 wow，“(”是 open，“|”是 bar，等等）

ASCII值	字符	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 表示。请注意，大写和小写字母恰好相差 32，这个观察可以用作检查或切换大小写的捷径。未显示的 ASCII 值包括格式化字符。例如，8 表示退格，9 表示“tab”字符，13 表示回车。另一个有用的值是 0 表示 null，C 语言用它来标记字符串结尾

例题 | ASCII 码对比二进制数

我们可以用一串 ASCII 码而不是整数来表示数。如果用 ASCII 码表示 10 亿这个数，相对于 32 位整数，会增加多少存储？

答案 | 十亿是 1 000 000 000，需要 10 个 ASCII 码表示，每个 8 位长。因此存储增长为（10 × 8）/ 32 倍[⊖]，即 2.5 倍。除了存储上的增长之外，硬件对这样的十进制数进行加减乘除也是困难的，并伴有更大的能耗。这些困难解释了为什么计算机专家变得相信二进制计算机是自然的，而偶尔出现的十进制计算机是奇怪的。

一系列指令可以从双字中提取一个字节，因此对双字的加载和存储足以传输字节和字。但由于某些程序中文本的流行，所以 RISC-V 提供了字节转移指令。*加载无符号字节*（lbu）指令从内存加载一个字节，将其放在寄存器的最右边 8 位。*存储字节*（sb）指令从寄存器的最右边 8 位取一个字节并将其写入内存。因此，我们复制一个字节的顺序如下：

```
lbu x12, 0(x10)    // Read byte from source
sb  x12, 0(x11)    // Write byte to destination
```

字符通常组合成具有可变数量的字符串。字符串的表示有三种选择：（1）字符串的第一个位置保留，用于给出字符串的长度；（2）附加带有字符串长度（如在结构体中）的变量；

⊖ 对比二进制表示。——译者注

（3）字符串的最后位置用一个字符标记字符串结尾。C 语言使用第三种选择，使用值为 0 的字节终止字符串（在 ASCII 中命名为 null）。因此，字符串“Cal”在 C 语言中用以下 4 个字节表示，十进制数表示为 67、97、108 和 0。（下面我们将看到，Java 使用第一个选项。）

例题 | 编译字符串复制程序，展示如何使用 C 语言字符串

strcpy 过程将字符串 y 复制到字符串 x，C 语言使用 null 字节标记字符串结束：

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

编译后的 RISC-V 汇编代码是什么？

答案 | 下面是基本的 RISC-V 汇编代码段。假设数组 x 和 y 的基址存放在 x10 和 x11 中，而 i 在 x19 中。strcpy 调整栈指针，然后将保存的寄存器 x19 保存在栈中：

```
strcpy:
    addi  sp, sp, -8       // adjust stack for 1 more item
    sd    x19, 0(sp)      // save x19
```

将 i 初始化为 0，下一条指令通过 0 加 0 将 x19 设为 0 并将结果放在 x19 中：

```
add  x19, x0, x0     // i = 0+0
```

这是循环的开始。y [i] 的地址首先通过将 i 加到 y [] 来形成：

```
L1: add  x5, x19, x11  // address of y[i] in x5
```

注意，我们不必将 i 乘以 8，因为 y 是字节数组而不是双字，如前面的例子中那样。

为了加载 y[i] 中的字符，我们使用无符号加载字节，将字符放入 x6 中：

```
lbu  x6, 0(x5)  // x6 = y[i]
```

类似的地址计算将 x [i] 的地址放在 x7 中，然后 x6 中的字符存储在那个地址中。

```
add    x7, x19, x10     // address of x[i] in x7
sb     x6, 0(x7)        // x[i] = y[i]
```

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

```
beq x6, x0, L2
```

如果不是，递增 i 继续循环：

```
addi  x19, x19, 1   // i = i + 1
jal   x0, L1        // go to L1
```

如果不继续循环，它就是字符串的最后一个字符；我们恢复 x19 和栈指针，然后返回。

```
L2: ld      x19, 0(sp)     // restore old x19
    addi    sp, sp, 8      // pop 1 doubleword off stack
    jalr    x0, 0(x1)     // return
```

在 C 语言中字符串复制通常使用指针而不是数组，以避免在上面的代码中对 i 进行操作。有关数组与指针的说明，请参见 2.14 节。

由于上面的 strcpy 过程是一个叶过程，编译器可以将 i 分配给临时寄存器并避免保存和恢复 x19。因此，我们可以将它们视为被调用者在方便的时候可使用的寄存器，而不是将这些寄存器视为临时寄存器。当编译器找到一个叶过程时，它会在使用必须保存的寄存器之前耗尽所有临时寄存器。

Java 中的字符和字符串

Unicode 是大多数人类语言中字母表的通用编码。图 2-16 给出了 Unicode 字母表的列表，Unicode 中的字母表几乎与 ASCII 中的有用符号一样多。为了更具包容性，Java 将 Unicode 用于字符。默认情况下，它使用 16 位来表示一个字符。

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 中的示例字母表。Unicode 版本 4.0 具有超过 160 个“块”，每个块是一个符号集合的名称。每个块都是 16 的倍数。例如，希腊语从 0370_{16} 开始，西里尔语从 0400_{16} 开始。前三列显示了 48 个块，这些块大致以 Unicode 的数字顺序对应于人类语言。最后一列 16 个块是多种语言的，且不按顺序。默认的是 16 位编码，称为 UTF-16。称为 UTF-8 的变长编码将 ASCII 子集保持为 8 位，并对其他字符使用 16 或 32 位。UTF-32 每个字符使用 32 位。想了解更多信息，请访问 www.unicode.org

RISC-V 指令系统具有加载和存储这种 16 位半字的指令。load half unsigned（加载无符号半字）从内存中读取一个半字，将它放在寄存器的最右边 16 位，用零填充最左边的 48 位。与加载字节一样，*加载半字*（lh）将半字视为有符号数，因此进行符号扩展以填充寄存器的最左边 48 位。*存储半字*（sh）从寄存器的最右边 16 位取半字并将其写入内存。我们按下面的序列来复制一个半字：

```
lhu x19, 0(x10) // Read halfword (16 bits) from source
sh  x19, 0(x11) // Write halfword (16 bits) to dest
```

字符串是标准的 Java 类，具有专门的内置支持和用于连接、比较和转换的预定义方法。与 C 语言不同，Java 包含一个给出字符串长度的字，类似于 Java 数组。

详细阐述 RISC-V 软件需要保持栈的“四字”（16 字节）地址对齐，以获得更好的性能。这意味着在栈上分配的 char 变量可能占用多达 16 个字节，即使它并不需要这么多。但是，C 字符串变量或字节型数组会把每 16 个字节压缩为“四字”，Java 字符串变量或 short 型数组将每 8 个半字压缩为“四字”。

详细阐述 为了反映网络的国际性，如今的大多数网页都使用 Unicode 而不是 ASCII。因此现在 Unicode 可能比 ASCII 更受欢迎。

详细阐述 RISC-V 还包括将 32 位值移入和移出存储器的指令。加载无符号字（lwu）将 32 位字从存储器加载到寄存器的最右边 32 位，用零填充最左边的 32 位。加载字（lw）用第 31 位的值填充最左边的 32 位。存储字（sw）从寄存器的最右边 32 位取一个字并将其存储到存储器中。

自我检测

I. 以下关于 C 和 Java 中字符和字符串的陈述哪些是正确的？

1. C 中的字符串占用的内存大约是 Java 中相同字符串的一半。
2. 字符串只是 C 和 Java 中一维字符数组的非正式名称。
3. C 和 Java 中的字符串使用 null（0）来标记字符串的结尾。
4. 对字符串的操作（如求长度）在 C 中比在 Java 中更快。

II. 以下哪种类型的变量在存放 $1\,000\,000\,000_{10}$ 时占用最多的内存空间？

1. C 语言中的 long long int
2. C 语言中的 string
3. Java 中的 string

2.10 对大立即数的 RISC-V 编址和寻址

虽然将所有 RISC-V 指令保持 32 位长可以简化硬件，但有时候使用 32 位或更大的常量或地址会很方便。本节从较大常量的一般解决方案开始，然后描述了分支指令中使用的指令地址优化。

2.10.1 大立即数

虽然常量通常很短并且适合 12 位字段，但有时它们也会更大。

RISC-V 指令系统包括指令 load upper immediate（取立即数高位，lui），用于将 20 位常数加载到寄存器的第 31 位到第 12 位。将第 31 位的值复制填充到最左边 32 位，最右边的 12 位用 0 填充。例如，这条指令允许使用两条指令创建 32 位常量。lui 使用新的指令格式——U 型，因为其他格式不能支持如此大的常量。

例题 加载一个 32 位常数

将以下 64 位常量加载到寄存器 x19 的 RISC-V 汇编代码是什么？

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

答案 首先，我们使用 lui 加载 12 到 31 位，十进制值为 976：

```
lui    x19, 976 // 976decimal = 0000 0000 0011 1101 0000
```

之后寄存器 x19 的值是：

```
0000000 00000000 00000000 00000000 00000000 00111101 00000000 00000000
```

下一步是添加最低 12 位，其十进制值为 1280：

```
addi    x19, x19, 1280     // 1280decimal = 00000101 00000000
```

寄存器 x19 中的最终值即是所需的：

```
00000000 00000000 00000000 00000000 00000000 00111101 00000101 00000000
```

详细阐述 在前面的例子中，常量的第 11 位为 0。如果第 11 位已经设为 1，则会出现额外的复杂情况：12 位立即数是符号扩展的，因此加数将为负数。这意味着除了添加常量的最右边 11 位之外，我们还需要减去 2^{12}。为了弥补这个错误，只需将 lui 加载的常量添加一个 1，因为 lui 常量缩小了 2^{12} 倍。

硬件/软件接口 编译器或汇编程序必须将大的常量分解为多个部分，然后将它们重新组装到寄存器中。正如你所料，对于加载和存储指令中的常量来说，立即数字段的大小限制是一个问题。

因此，RISC-V 机器语言的符号表示不再受硬件限制，而是受限于汇编程序的构建者选择包含的内容（参见 2.12 节）。我们坚持以靠近硬件层次的方式来解释计算机的体系结构，注意，当我们使用汇编程序的扩展语言时，在实际处理器的实现中是找不到的。

2.10.2　分支中的寻址

RISC-V 分支指令使用称为 SB 型的 RISC-V 指令格式。这种格式可以表示从 −4096 到 4094 的分支地址，以 2 的倍数表示。由于最近的一些原因，它只能跳转到偶数地址。SB 型格式包括一个 7 位操作码、一个 3 位功能码、两个 5 位的寄存器操作数（rs1 和 rs2）和一个 12 位地址立即数。该地址使用特殊的编码方式，简化了数据通路设计，但使组装变得复杂。下面这条指令

```
bne  x10, x11, 2000  // if x10 != x11, go to location 2000ten = 0111 1101 0000
```

可以组装为这种格式（正如我们将看到的，它实际上有点复杂）：

0	111110	01011	01010	001	1000	0	1100111
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode

其中条件分支的操作码是 1100111_2，而 bne 的 funct3 码是 001_2。

无条件跳转 – 链接指令（jal）是唯一使用 UJ 型格式的指令。该指令由一个 7 位操作码、一个 5 位目标寄存器操作数（rd）和一个 20 位地址立即数组成。链接地址，即 jal 之后的指令的地址，被写入 rd 中。

与 SB 型格式一样，UJ 型格式的地址操作数使用特殊的立即数编码方式，它不能编码奇数地址。所以，

```
jal  x0, 2000  // go to location 2000ten = 0111 1101 0000
```

被组装为这种格式：

0	1111101000	0	00000000	00000	1101111
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode

如果程序的地址必须适合这个 20 位字段，则意味着没有程序可能大于 2^{20}，而这对于今

天的需求来说太小，不是一个现实的选择。另一种方法是指定一个与分支地址偏移量相加的寄存器，以便分支指令可以按如下来计算：

$$\text{程序计数器} = \text{寄存器内容} + \text{分支地址偏移量}$$

这样就允许程序大到 2^{64}，并且仍然能够使用条件分支指令，解决了分支地址大小问题。那么问题是使用哪个寄存器？

答案来自于如何使用条件分支指令。条件分支指令在循环和 if 语句中使用，因此它们倾向于转移到附近的指令。例如，SPEC 基准测试中约有一半的条件分支跳到小于 16 条指令距离的位置。由于程序计数器（PC）包含当前指令的地址，如果我们使用 PC 作为该寄存器，可以在距离当前指令的 $\pm 2^{10}$ 个字的地方分支，或者跳转到距离当前指令 $\pm 2^{18}$ 个字的地方。几乎所有循环和 if 语句都小于 2^{10} 个字，因此 PC 是理想的选择。这种形式的寻址方式称为 **PC 相对寻址**。

PC 相对寻址：一种寻址方式，它的地址是 PC 和指令中的常量之和。

与最新的计算机一样，RISC-V 对条件分支和无条件跳转使用 PC 相对寻址，因为这些指令的目标地址可能距离分支很近。另一方面，过程调用可能需要转移超过 2^{18} 个字的距离，因为不能保证被调用者接近调用者。因此，RISC-V 允许使用双指令序列来非常长距离地跳转到任何 32 位地址：`lui` 将地址的第 12 位至第 31 位写入临时寄存器，`jalr` 将地址的低 12 位加到临时寄存器并跳转到目标位置。

由于 RISC-V 指令长度为 4 个字节，因此 RISC-V 分支指令可以设计为通过让 PC 相对偏移表示分支和目标指令之间的字数而不是字节数，以便扩展其范围。但是，RISC-V 架构师也希望支持只有 2 个字节长的指令，因此 PC 相对偏移表示分支和目标指令之间的半字数。因此，`jal` 指令中的 20 位地址字段可以编码为距当前 PC $\pm 2^{19}$ 个半字或 ± 1 MiB 的距离。类似地，条件分支指令中的 12 位立即数字段也是半字地址，这意味着它表示 13 位的字节地址。

例题 | 描述机器语言中的分支偏移

2.7.1 节的 while 循环已编译为下列 RISC-V 汇编代码：

```
Loop:slli x10, x22, 3     // Temp reg x10 = i * 8
     add  x10, x10, x25   // x10 = address of save[i]
     ld   x9, 0(x10)      // Temp reg x9 = save[i]
     bne  x9, x24, Exit   // go to Exit if save[i] != k
     addi x22, x22, 1     // i = i + 1
     beq  x0, x0, Loop    // go to Loop
Exit:
```

如果我们假设将循环的开始放在内存的 80000 处中，那么这个循环的 RISC-V 机器代码是什么？

答案 | 汇编指令及其地址如下：

地址	指令					
80000	0000000	00011	10110	001	01010	0010011
80004	0000000	11001	01010	000	01010	0110011
80008	0000000	00000	01010	011	01001	0000011
80012	0000000	11000	01001	001	01100	1100011
80016	0000000	00001	10110	000	10110	0010011
80020	1111111	00000	00000	000	01101	1100011

请记住，RISC-V 指令是字节地址，因此相连字的地址相差 4。第四行上的 `bne` 指令将

3 个字或 12 个字节加到指令地址上，指明分支目标和分支指令（12 + 80012）相关，而不使用完整的目标地址（80024）。最后一行上的分支指令对向后分支（−20 + 80020）进行类似的计算，对应于标签 Loop。

硬件/软件接口 大多数条件分支到达附近的位置，但偶尔会转移到很远的位置，远远超过条件分支指令中的 12 位地址能表示的范围。汇编程序解决该问题就如同处理大地址或常量的方法一样：插入无条件跳转到分支目标，并将条件取反，以便条件分支决定是否跳过该无条件跳转。

例题 | 远距离分支

寄存器 x10 等于 0 时给定一个分支，

```
beq    x10, x0, L1
```

用一对提供更大分支距离的指令替换它。

答案 | 用下面的指令替换短地址条件分支指令：

```
      bne      x10, x0, L2
      jal      x0, L1
L2:
```

2.10.3 RISC-V 寻址模式总结

多种不同的寻址形式通常称为**寻址模式**。图 2-17 显示了每种寻址模式如何识别操作数。RISC-V 指令的寻址模式如下：

寻址模式： 根据对操作数和（或）地址使用的不同，在多种寻址方式中加以区分的寻址机制。

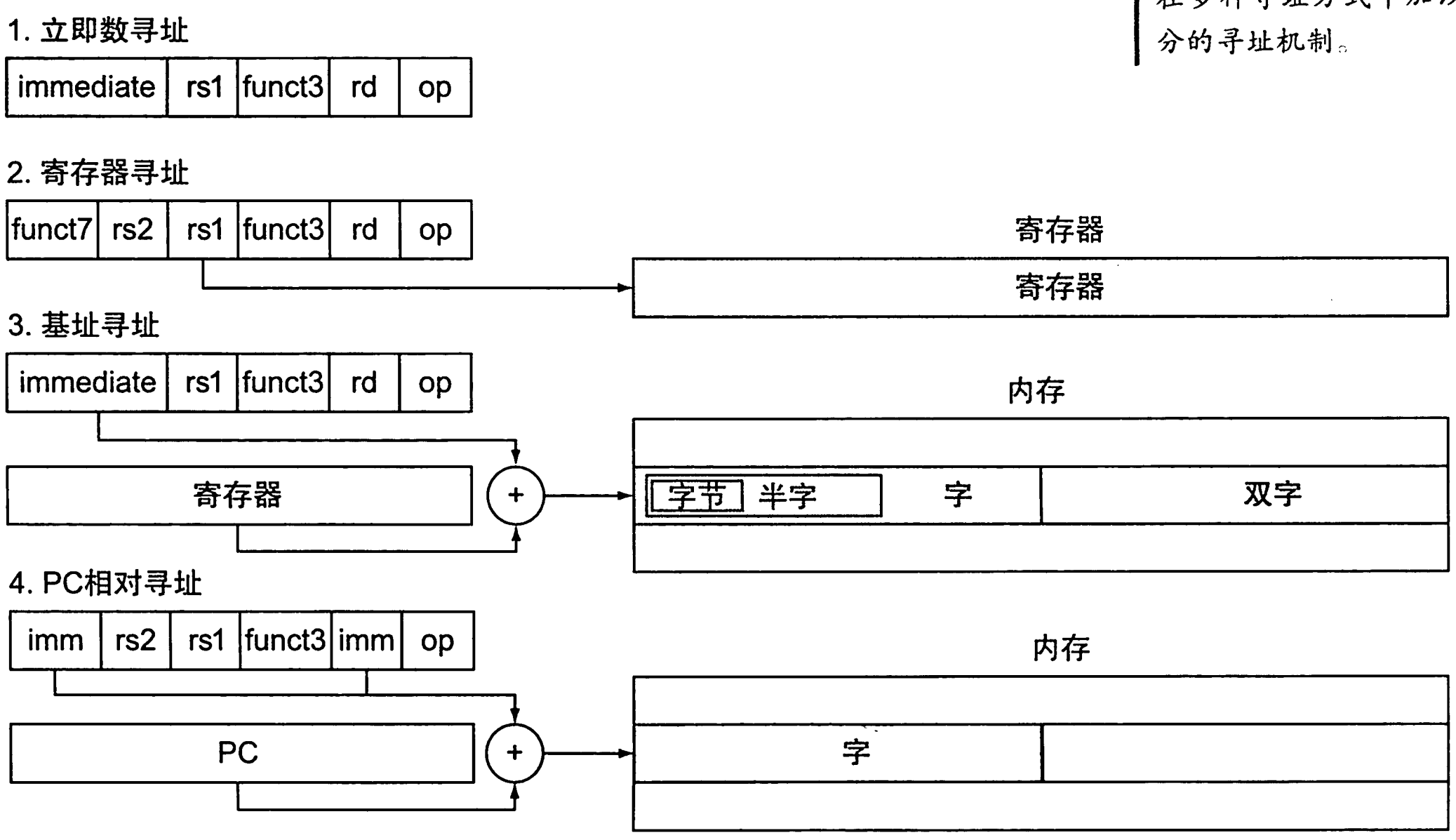

图 2-17　四种 RISC-V 寻址模式的示意图。操作数以灰色阴影表示。寻址模式 3 的操作数在内存中，而模式 2 的操作数在寄存器中。注意加载和存储对字节、半字、字或双字的访问有不同的版本。对于寻址模式 1，操作数是指令本身的一部分。模式 4 寻址指令在内存中，将长地址与 PC 相加。注意一种操作可以使用多个寻址模式。例如，加法可以使用立即数寻址（addi）和寄存器寻址（add）