1. 立即数寻址，操作数是指令本身的常量。
2. 寄存器寻址，操作数在寄存器中。
3. 基址或偏移寻址，操作数于内存中，其地址是寄存器和指令中的常量之和。
4. PC 相对寻址，分支地址是 PC 和指令中常量之和。

2.10.4 机器语言译码

有时必须通过逆向工程将机器语言恢复到初始的汇编语言。例如发生“内存转储”（core dump）时。图 2-18 显示了 RISC-V 机器语言对应的二进制编码。这个图有助于在汇编语言和机器语言之间进行手动翻译。

格式	指令	操作码	funct3	funct7
R型	add	0110011	000	0000000
	sub	0110011	000	0100000
	sll	0110011	001	0000000
	xor	0110011	100	0000000
	srl	0110011	101	0000000
	sra	0110011	101	0000000
	or	0110011	110	0000000
	and	0110011	111	0000000
	lr.d	0110011	011	0001000
	sc.d	0110011	011	0001100
I型	lb	0000011	000	n.a.
	lh	0000011	001	n.a.
	lw	0000011	010	n.a.
	ld	0000011	011	n.a.
	lbu	0000011	100	n.a.
	lhu	0000011	101	n.a.
	lwu	0000011	110	n.a.
	addi	0010011	000	n.a.
	slli	0010011	001	000000
	xori	0010011	100	n.a.
	srli	0010011	101	000000
	srai	0010011	101	010000
	ori	0010011	110	n.a.
	andi	0010011	111	n.a.
	jalr	1100111	000	n.a.
S型	sb	0100011	000	n.a.
	sh	0100011	001	n.a.
	sw	0100011	010	n.a.
	sd	0100011	111	n.a.
SB型	beq	1100111	000	n.a.
	bne	1100111	001	n.a.
	blt	1100111	100	n.a.
	bge	1100111	101	n.a.
	bltu	1100111	110	n.a.
	bgeu	1100111	111	n.a.
U型	lui	0110111	n.a.	n.a.
UJ型		1101111	n.a.	n.a.

图 2-18 RISC-V 指令的编码。所有指令都有一个操作码字段，除了 U 型和 UJ 型之外的所有格式都使用 funct3 字段。R 型指令使用 funct7 字段，立即数移位（slli，srli，srai）使用 funct6 字段

例题 | 机器码译码

与下面这条机器指令对应的汇编语言语句是什么？

00578833_{16}

答案 | 第一步是将十六进制转换为二进制：

0000 0000 0101 0111 1000 1000 0011 0011

要知道如何解释这些位，我们需要确定指令格式，为此首先需要确定操作码。操作码是最右边的 7 位，即 0110011。在图 2-18 中搜索该值，我们看到操作码对应于 R 型算术指令。因此，我们可以将二进制格式解析为下图中列出的字段：

funct7	rs2	rs1	funct3	rd	opcode
0000000	00101	01111	000	10000	0110011

我们通过查看字段值来译码指令的剩余部分。funct7 和 funct3 字段均为零，表示指令是加法。操作数寄存器 rs2 字段的十进制值为 5，rs1 为 15，rd 为 16。这些数字代表寄存器 x5、x15 和 x16。现在我们可以得到汇编指令：

```
add x16, x15, x5
```

图 2-19 显示了所有 RISC-V 指令格式。图 2-1 显示了本章中介绍的 RISC-V 汇编语言。下一章将介绍用于实数的乘法、除法和算术的 RISC-V 指令。

名称 （字段大小）	字段						备注
	7 位	5 位	5 位	3 位	5 位	7 位	
R型	funct7	rs2	rs1	funct3	rd	opcode	算术指令格式
I型	immediate[11:0]		rs1	funct3	rd	opcode	加载&立即数算术
S型	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	存储
SB型	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	条件分支格式
UJ型	immediate[20,10:1,11,19:12]				rd	opcode	无条件跳转
U型	immediate[31:12]				rd	opcode	大立即数格式

图 2-19　RISC-V 指令格式

自我检测

I. RISC-V 中条件分支的字节地址范围是多少（K = 1024）？

1. 地址在 0 到 4K−1 之间
2. 地址在 0 到 8K−1 之间
3. 分支前后地址范围各大约为 2K
4. 分支前后地址范围各大约为 4K

II. RISC-V 中跳转 – 链接指令的字节地址范围是多少（M = 1024K）？

1. 地址在 0 到 512K−1 之间
2. 地址在 0 到 1M−1 之间
3. 分支前后地址范围各大约为 512K
4. 分支前后地址范围各大约为 1M

2.11 指令与并行性：同步

当任务之间相互独立时，并行执行更为容易，但通常任务之间需要协作。协作通常意味着一些任务正在写入其他任务必须读取的值。需要知道任务何时完成写入以便其他任务安全地读出，因此任务之间需要同步。如果它们不同步，则存在**数据竞争**（data race）的危险，那么程序的结果会根据事件发生的次序而改变。

数据竞争：如果来自两个不同的线程的访存请求访问同一个位置，至少有一个是写，且连续出现，那么这两次存储访问形成了数据竞争。

例如，回想一下 1.8 节中提到的 8 个记者写作一篇故事的类比。假设一位记者在写完总结之前需要阅读前面所有的章节。那么，他必须知道其他记者什么时候完成各自的章节，以便之后不会出现他写完总结后其他记者又修改了各自章节的情况。也就是说，他们最好同步每个部分的写作和阅读，以便与前面章节的内容一致。

在计算中，同步机制通常由用户级的软件例程所构建，而这依赖于硬件提供的同步指令。在本节中，我们将重点介绍加锁（lock）和解锁（unlock）同步操作的实现。加锁和解锁可直接用于创建只有单个处理器可以操作的区域，称为互斥（mutual exclusion）区，以及实现更复杂的同步机制。

在多处理器中实现同步所需的关键是一组硬件原语，能够提供以原子方式读取和修改内存单元的能力。也就是说，在内存单元的读取和写入之间不能插入其他任何操作。如果没有这样的能力，构建基本同步原语的成本将会很高，并会随着处理器数量的增加而急剧增加。

有许多基本硬件原语的实现方案，所有这些都提供了原子读和原子写的能力，以及一些判断读写是否是原子操作的方法。通常，体系结构设计人员不希望用户使用基本的硬件原语，而是期望系统程序员使用原语来构建同步库，这个过程通常复杂且棘手。

我们从**原子交换**（atomic exchange 或 atomic swap）原语开始，展示如何使用它来构建基本同步原语。它是构建同步机制的一种典型操作，它将寄存器中的值与存储器中的值进行交换。

为了了解如何使用它来构建基本同步原语，假设要构建一个简单的锁变量，其中值 0 用于表示锁变量可用，值 1 用于表示锁变量已被占用。处理器尝试通过将寄存器中的 1 与该锁变量对应的内存地址的值进行交换来设置加锁。如果某个其他处理器已声明访问该锁变量，则交换指令的返回值为 1，表明该锁已被其他处理器占用，否则为 0，表示加锁成功。在后一种情况下，锁变量的值变为 1，以防止其他处理器也加锁成功。

例如，考虑两个处理器尝试同时进行交换操作：这种竞争会被阻止，因为其中一个处理器将首先执行交换，并返回 0，而第二个处理器在进行交换时将返回 1。使用交换原语实现同步的关键是操作的原子性：交换是不可分割的，硬件将对两个同时发生的交换进行排序。尝试以这种方式设置同步变量的两个处理器都不可能认为它们同时设置了变量。

实现单个的原子存储操作为处理器的设计带来了一些挑战，因为它要求在单条不可中断的指令中完成存储器的读和写操作。

另一种方法是使用指令对，其中第二条指令返回一个值，该值表示该指令对是否被原子执行。如果任何处理器执行的所有其他操作都发生在该对指令之前或之后，则该指令对实际上是原子的。因此，当指令对实际上是原子操作时，没有其他处理器可以在指令对之间改变值。

在 RISC-V 中，这对指令指的是一个称为保留加载（load-reserved）双字（`lr.d`）的特殊

加载指令和一个称为条件存储（store-conditional）双字（sc.d）的特殊存储指令。这些指令按序使用：如果保留加载指令指定的内存位置的内容在条件存储指令执行到同一地址之前发生了变化，则条件存储指令失败且不会将值写入内存。条件存储指令定义为将（可能是不同的）寄存器的值存储在内存中，如果成功则将另一个寄存器的值更改为0，如果失败则更改为非零值。因此，sc.d指定了三个寄存器：一个用于保存地址，一个用于指示原子操作失败或成功，还有一个用于如果成功则将值存储在内存中。由于保留加载指令返回初始值，并且条件存储指令仅在成功时返回0，因此以下序列在寄存器x20中指定的内存位置上实现原子交换：

```
again:lr.d x10, (x20)          // load-reserved
      sc.d x11, x23, (x20)     // store-conditional
      bne  x11, x0, again      // branch if store fails
      addi x23, x10, 0         // put loaded value in x23
```

每当处理器干预并修改lr.d和sc.d指令之间的内存中的值时，sc.d就会将非零值写入x11，从而导致代码序列重新执行。在此序列结束时，x23的值和x20指向的内存位置的值发生了原子交换。

详细阐述 虽然同步是为多处理器而提出的，但原子交换对于单个处理器操作系统中处理多个进程也很有用。为了确保单个处理器中的执行不受任何干扰，如果处理器在两个指令对之间进行上下文切换，则条件存储也会失败（参见第5章）。

详细阐述 保留加载/条件存储机制的一个优点是可以用于构建其他同步原语，例如原子的比较和交换（atomic compare and swap）或原子的取后加（atomic fetch-and-increment），这在一些并行编程模型中使用。这些同步原语的实现需要在lr.d和sc.d之间插入更多指令，但不会太多。

由于条件存储会在另一个store尝试加载保留地址或异常之后失败，因此必须注意选择在两个指令之间插入哪些指令。特别是，只有保留加载/条件存储块中的整点算术、前向分支和后向分支被允许执行且不会出现问题；否则，可能会产生死锁情况——由于重复的页错误，处理器永远无法完成sc.d。此外，保留加载和条件存储之间的指令数应该很少，以将由于不相关事件或竞争处理器导致条件存储频繁失败的可能性降至最低。

详细阐述 虽然上面的代码实现了原子交换，但下面的代码可以更有效地获取寄存器x20对应存储中的锁变量，其中值0表示锁变量是空闲的，值1表示锁变量被占用：

```
        addi x12, x0, 1       // copy locked value
again: lr.d  x10, (x20)       // load-reserved to read lock
       bne   x10, x0, again   // check if it is 0 yet
       sc.d x11, x12, (x20) // attempt to store new value
       bne  x11, x0, again  // branch if store fails
```

我们只使用普通的存储指令将0写入该位置来释放锁变量：

```
sd x0, 0(x20)     // free lock by writing 0
```

自我检测 什么时候使用类似保留加载和条件存储的原语？

1. 当并行程序中相互协作的线程需要同步以获得用于读取和写入共享数据的正确行为时。
2. 当单处理器上相互协作的进程需要同步以读取和写入共享数据时。

2.12 翻译并启动程序

本节介绍了将存储（磁盘或闪存）文件中的 C 程序转换为计算机上可运行程序的四个步骤。图 2-20 显示了转换的层次结构。有些系统将这些步骤结合起来以减少转换时间，但程序的转换过程一定会经历这四个逻辑阶段。本节遵循此转换层次结构。

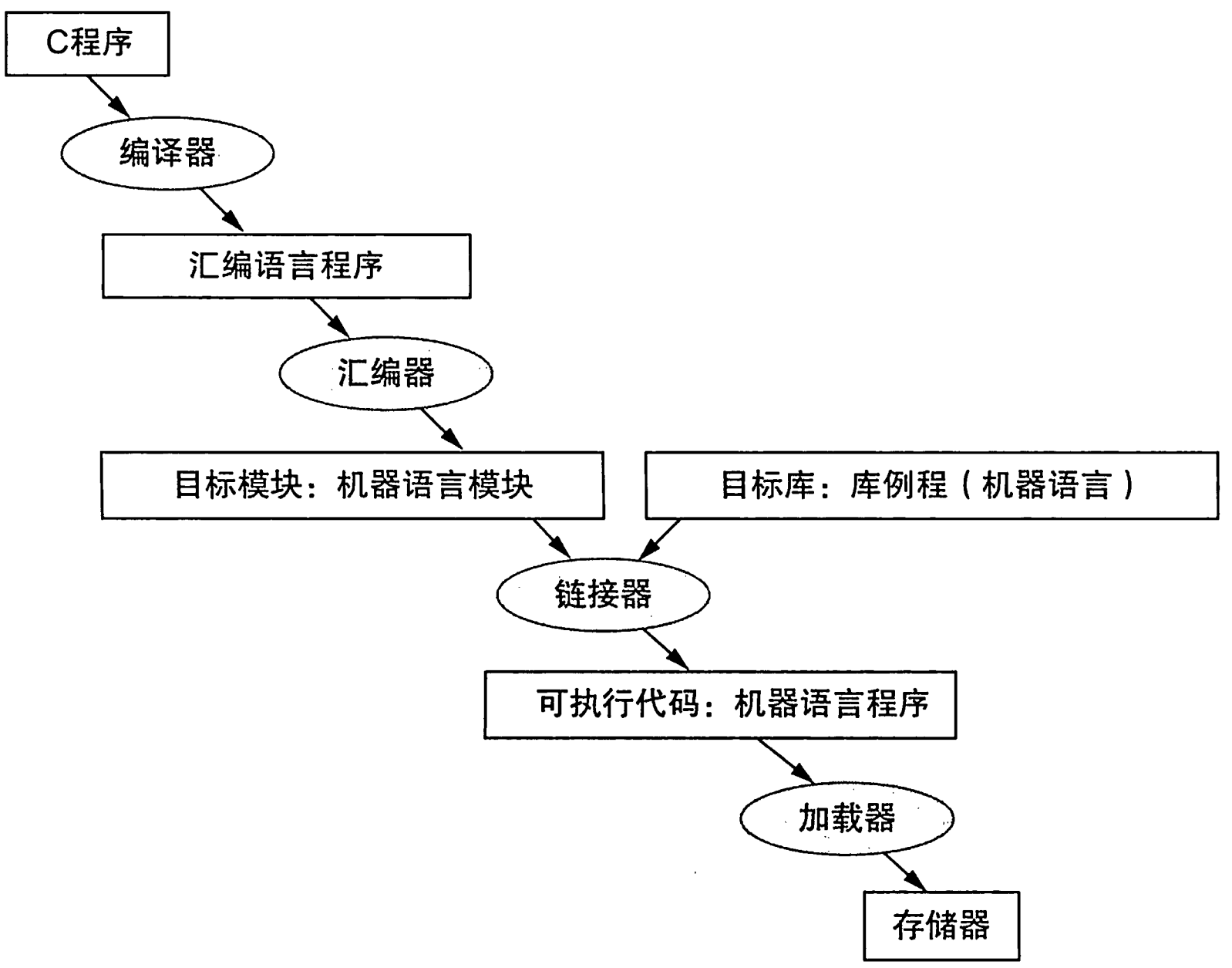


图 2-20 C 语言的转换层次结构。首先将高级语言程序编译成汇编语言程序，然后用机器语言组装成目标模块。链接器将多个模块与库程序组合在一起以解析所有引用。然后，加载器将机器代码放入适当的存储器位置以供处理器执行。为了加快转换过程，可以跳过或将一些步骤组合到一起。一些编译器直接生成目标模块，一些系统使用链接加载器执行最后两个步骤。为了识别文件类型，UNIX 遵循文件的后缀约定：C 源文件命名为 x.c，汇编文件命名为 x.s，目标文件命名为 x.o，静态链接库程序为 x.a，动态链接库路径为 x.so，以及默认情况下，可执行文件称为 a.out。MS-DOS 使用后缀 .C、.ASM、.OBJ、.LIB、. DLL 和 .EXE 的效果相同

2.12.1 编译器

编译器将 C 程序转换为机器能理解的符号形式——汇编语言程序（assembly language program）。高级语言程序比汇编语言使用更少的代码行，因此程序员的工作效率更高。

汇编语言：一种能被翻译为二进制机器语言的符号语言。

在 1975 年，许多操作系统和汇编器都是用**汇编语言**（assembly language）编写的，因为那时的计算机内存很小而且编译器效率还很低下。如今每个 DRAM 芯片的内存容量增加了数百万倍，从而减少了程序员对程序大小的关注，今天的优化编译器几乎可以像汇编语言专家那样生成汇编语言程序，而且对大型程序的优化效果有时甚至比人工优化更好。

2.12.2 汇编器

由于汇编语言是高层软件的接口，因此汇编器还可以处理机器指令的常见变体，就像这

些变体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出现简化了程序转换和编程。这类指令称为**伪指令**。

伪指令：一种汇编指令的常见变体，可以把它看作汇编语言指令。

如上所述，RISC-V 硬件确保寄存器 x0 总是取 0。也就是说，每当使用寄存器 x0 时，它提供 0，如果程序员尝试更改 x0 中的值，则新值会被直接丢弃。寄存器 x0 用于创建汇编语言指令，将一个寄存器的内容复制到另一个寄存器。因此，即使在 RISC-V 机器语言中不存在这条指令，RISC-V 汇编器也能够识别以下指令：

```
li x9, 123      // load immediate value 123 into register x9
```

汇编器将此汇编语言指令转换为与以下指令等效的机器语言：

```
addi x9, x0, 123 // register x9 gets register x0 + 123
```

RISC-V 汇编器还将 mv（move）转换为 addi 指令。从而

```
mv x10, x11  // register x10 gets register x11
```

变为

```
addi x10, x11, 0  // register x10 gets register x11 + 0
```

汇编器还接受 j Label 作为 jal x0, Label 的替代，无条件跳转到标签位置。它还将跳转到远距离的分支指令转换为一个分支指令和一个跳转指令。如上所述，RISC-V 汇编器允许将大常量加载到寄存器中，尽管立即数指令的位数有限。因此，上面介绍的 load immediate（li）伪指令可以创建大于 addi 的立即数字段可包含的常量；加载地址（la）宏对符号地址的工作方式类似。最后，它可以通过确定程序员想要的指令变体来简化指令系统。例如，算术和逻辑指令使用常量时，RISC-V 汇编器不要求程序员指定指令的立即数版本，它只是生成正确的操作码。从而将

```
and x9, x10, 15 // register x9 gets x10 AND 15
```

变为

```
andi x9, x10, 15 // register x9 gets x10 AND 15
```

我们在指令中包含“i”以提醒读者，andi 与没有立即数操作数的 and 指令不同，具有不同的指令格式，会生成不同的操作码。

总之，伪指令为 RISC-V 提供了比硬件实现更丰富的汇编语言指令系统。如果要编写汇编程序，请使用伪指令来简化任务。但是，要了解 RISC-V 体系结构并确保获得最佳性能，请学习图 2-1 和图 2-18 中真正的 RISC-V 指令。

汇编器也会接收不同基数的数字。除了二进制和十进制之外，它们通常接收比二进制更简短又很容易转换为位模式的基数。RISC-V 汇编器使用十六进制和八进制。

这种特性非常方便，但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为目标文件（object file），该目标文件是机器指令、数据和将指令正确放入内存所需信息的组合。

为了在汇编语言程序中产生每条指令的二进制版本，汇编器必须确定与所有标签相对应的地址。汇编器会跟踪分支中使用的标签和**符号表**中的数据传输指令。正如你所料，该表由符号和对应地址成对组成。

符号表：用于匹配标签名和指令所在内存的地址的表。

UNIX 系统的目标文件通常包含六个不同的部分：

- 目标文件头，描述了目标文件的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 允许程序使用静态数据，它在整个程序中都存在；也允许使用动态数据，它可以根据程序的需要增长或缩小。见图 2-13。）
- 重定位信息，标记了在程序加载到内存时依赖于绝对地址的指令和数据。
- 符号表，包含剩余的未定义的标签，例如外部引用。
- 调试信息，包含有关如何编译目标模块的简明描述，以便调试器可以将机器指令与 C 源文件相关联并使数据结构可读。

下一小节将介绍如何链接已汇编完成的子程序，例如库程序。

2.12.3 链接器

到目前为止我们所呈现的内容表明，对一个程序的一行进行单一更改需要编译和汇编整个程序。完全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来说尤其浪费，因为程序员将要编译和汇编根据定义几乎永远不会改变的例程。另一种方法是独立编译和汇编每个过程，因此更改一行代码只需要编译和汇编一个过程。这种替代方案需要一个新的系统程序，称为**链接编辑器**或**链接器**，它将所有独立汇编的机器语言程序“缝合”在一起。链接器有用的原因是修正代码要比重新编译和重新汇编快得多。

链接器：也叫链接编辑器，是一个系统程序，它将独立汇编的机器语言程序组合起来，并解析所有未定义的标签，最终生成可执行文件。

链接器的工作有三个步骤：

1. 将代码和数据模块按符号特征放入内存。
2. 决定数据和指令标签的地址。
3. 修正内部和外部引用。

链接器使用每个对象模块中的重定位信息和符号表来解析所有未定义的标签。这些引用发生在分支指令和数据地址中，因此该程序的工作与编辑器的工作非常相似：它找到旧地址并用新地址替换它们。“编辑器”是“链接编辑器”或简称“链接器”的原始名称。

如果解析了所有外部引用，则链接器接下来将确定每个模块将占用的内存位置。回想一下，图 2-13 显示了将程序和数据分配给内存的 RISC-V 准则。由于文件是单独汇编的，因此汇编器无法知道模块的指令和数据相对于其他模块的位置。当链接器将模块放入内存时，必须重定位所有的绝对引用（即与寄存器无关的内存地址）以反映其真实地址。

可执行文件：一种具有目标文件格式的功能程序，不包含未解析的引用。它可以包含符号表和调试信息。“剥离的可执行文件”不包含这些信息，可以包括用于加载器的重定位信息。

链接器生成可在计算机上运行的**可执行文件**。通常，此文件具有与目标文件相同的格式，但它不包含任何未解析的引用。具有部分链接的文件是可能的，例如库程序，在目标文件中仍然有未解析的地址。

例题 | 目标文件的链接

链接下面的两个目标文件。给出已完成的可执行文件的前几条指令的更新地址。用汇编

语言显示指令只是为了使例子利于理解；实际上，指令应是数字形式。

请注意，在目标文件中，我们用灰色表示链接过程中必须更新的地址和符号：引用过程A和B的地址的指令，以及引用数据双字X和Y的地址的指令。

目标文件首部			
	名字	过程A	
	正文大小	100_{16}	
	数据大小	20_{16}	
正文段	地址	指令	
	0	ld x10, 0(x3)	
	4	jal x1, 0	
	...	...	
数据段	0	(X)	
	...	...	
重定位信息	地址	指令类型	依赖
	0	ld	X
	4	jal	B
符号表	标签	地址	
	X	-	
	B	-	
	名字	过程B	
	正文大小	200_{16}	
	数据大小	30_{16}	
正文段	地址	指令	
	0	sd x11, 0(x3)	
	4	jal x1, 0	
	...	...	
数据段	0	(Y)	
	...	...	
重定位信息	地址	指令类型	依赖
	0	sd	Y
	4	jal	A
符号表	标签	地址	
	Y	-	
	A	-	

答案 | 过程A需要找到load指令中的可变标签X的地址，并找到jal指令中过程B的地址。过程B需要找到store指令中的可变标签Y的地址，及其jal指令中过程A的地址。

从图2-14我们知道正文段从地址0000 0000 0040 0000_{16}开始，数据段从0000 0000 1000 0000_{16}开始。过程A的正文放在第一个地址，数据放在第二个地址。过程A的目标文件首部表示其正文为100_{16}字节，数据为20_{16}字节，因此过程B正文的起始地址为40 0100_{16}，数据的起始地址是1000 0020_{16}。

可执行文件首部		
	正文大小	300_{16}
	数据大小	50_{16}
正文段	地址	指令
	0000 0000 0040 0000_{16}	ld x10, 0(x3)
	0000 0000 0040 0004_{16}	jal x1, 252

	...	...
	$0000\ 0000\ 0040\ 0100_{16}$	sd x11, 32(x3)
	$0000\ 0000\ 0040\ 0104_{16}$	jal x1, -260_{10}
	...	...
数据段	地址	
	$0000\ 0000\ 1000\ 0000_{16}$	(X)
	...	...
	$0000\ 0000\ 1000\ 0020_{16}$	(Y)
	...	...

现在链接器更新指令的地址字段。它使用指令类型字段来获得要编辑的地址格式。此处有三种类型：

1. 跳转和链接指令使用PC相对寻址。因此，对于地址 $40\ 0004_{16}$ 处的 jal 转到 $40\ 0100_{16}$（过程 B 的地址），它必须在其地址字段中放入（$40\ 0100_{16}-40\ 0004_{16}$）或 252_{10}。同样，因为 $40\ 0000_{16}$ 是过程 A 的地址，在 $40\ 0104_{16}$ 处的 jal 在其地址字段中放入负数 -260_{10}（$40\ 0000_{16}-40\ 0104_{16}$）。

2. load 指令的地址更为复杂，因为它们与基址寄存器有关。此示例使用 x3 作为基址寄存器，假设它初始化为 $0000\ 0000\ 1000\ 0000_{16}$。为了获得地址 $0000\ 0000\ 1000\ 0000_{16}$（双字 X 的地址），我们在地址 $40\ 0000_{16}$ 的 ld 的地址字段中放入 0_{10}。类似地，我们在地址 $40\ 0100_{16}$ 的 sd 的地址字段中放入 20_{16} 以获得地址 $0000\ 0000\ 1000\ 0020_{16}$（双字 Y 的地址）。

3. store 指令地址的处理方式与 load 指令类似，只是它们的 S 型指令格式与 load 指令的 I 型格式不同。我们在地址 $40\ 0100_{16}$ 的 sd 的地址字段中放入 32_{10}，得到地址 $0000\ 0000\ 1000\ 0020_{16}$（双字 Y 的地址）。

2.12.4 加载器

现在可执行文件在磁盘上，操作系统将其读取到内存并启动它。**加载器**在 UNIX 系统中遵循以下步骤：

1. 读取可执行文件首部以确定正文段和数据段的大小。
2. 为正文和数据创建足够大的地址空间。
3. 将可执行文件中的指令和数据复制到内存中。
4. 将主程序的参数（如果有）复制到栈顶。
5. 初始化处理器寄存器并将栈指针指向第一个空闲位置。
6. 跳转到启动例程，将参数复制到参数寄存器中并调用程序的主例程。当主例程返回时，启动例程通过 exit 系统调用终止程序。

加载器：将目标程序放在主存中以准备执行的系统程序。

2.12.5 动态链接库

本节首先描述了在程序运行之前链接库文件的传统方法。虽然这种静态方法是调用库例程的最快方法，但有一些缺点：

- 库例程成为可执行代码的一部分。如果发布了修复错误或支持新硬件设备的新版本库，则静态链接程序仍将继续使用旧版本。
- 它会加载执行过程中在任何位置可能会调用的所有库的所有例程，即使有些例程不一

事实上，计算机科学中的每一个问题都可以通过增加一个间接的中间层来解决。
David Wheeler

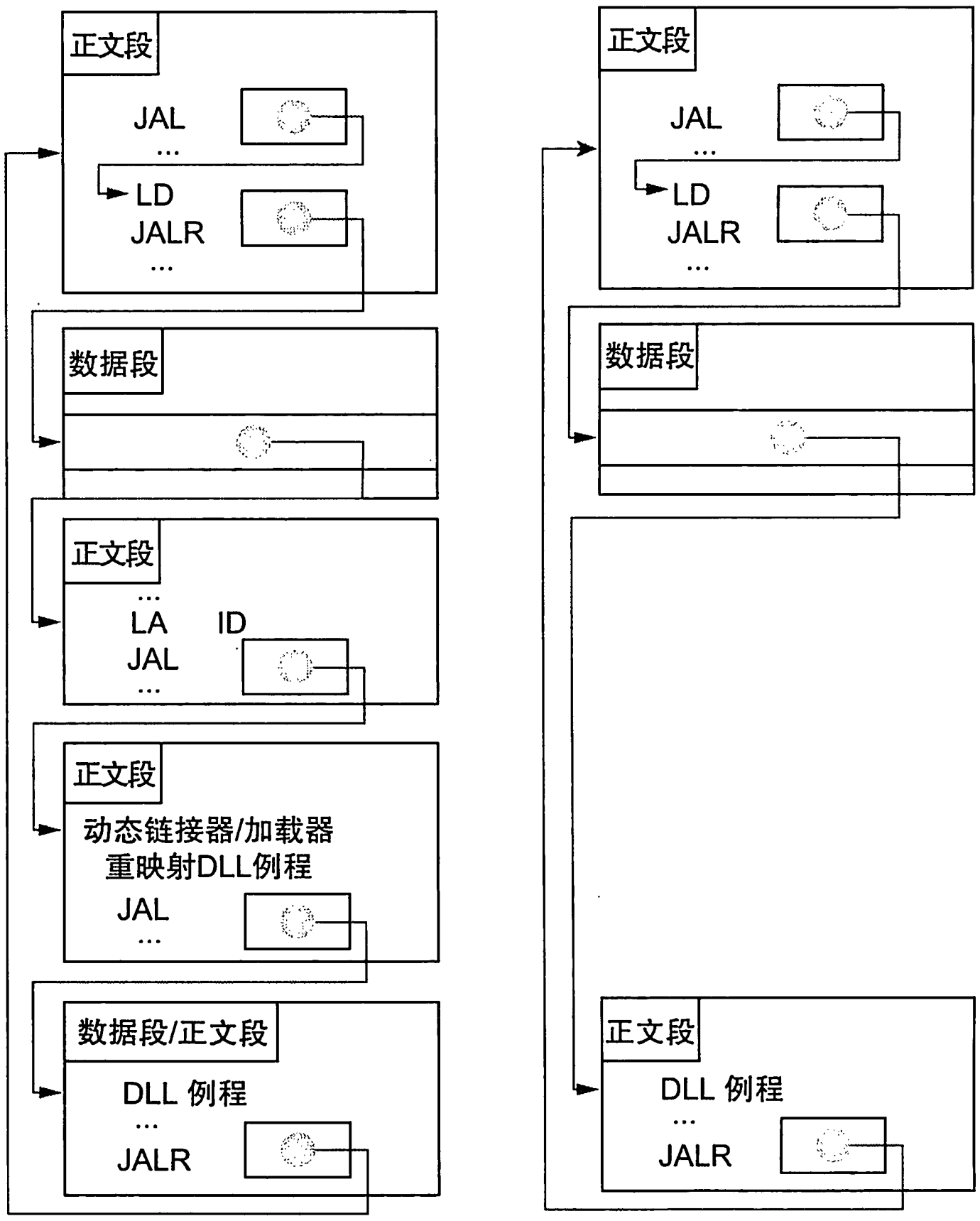
定会用到。相对于程序，库可能很大，例如，运行在 Linux 操作系统上的 RISC-V 系统的标准 C 库是 1.5 MiB。

这些缺点引出了**动态链接库**（dynamically linked libraries，DLL），其库例程在程序运行之前不会被链接和加载。程序和库例程都保存有关非局部过程及其名字的额外信息。在 DLL 的最初版本中，加载器运行一个动态链接程序，使用文件中的额外信息来查找相应的库并更新所有外部引用。

动态链接库：在执行期间链接到程序的库例程。

DLL 初始版本的缺点是，它仍然链接了可能被调用的库的所有例程，而不是在程序运行期间调用的那些例程。这种观察引出了 DLL 的延迟过程链接版本，其中每个例程仅在被调用之后才链接。

像这个领域的许多创新一样，这项技巧依赖于一个间接层次。图 2-21 展示了这种技术。它从非局部例程开始，在程序结束时调用一组虚例程，每个非局部例程有一个入口。每个虚入口都包含一个间接跳转。



a）对DLL例程的第一次调用　　b）对DLL例程的后续调用

图 2-21　通过延迟过程链接动态链接库。a）第一次调用 DLL 例程的步骤。b）在后续调用中跳过查找例程、重映射例程和链接例程的步骤。正如我们将在第 5 章中看到的那样，操作系统可以通过使用虚拟内存管理对其进行重映射来避免复制所需的例程

第一次调用库例程时，程序调用虚入口并执行间接跳转。这个跳转指向一段代码，它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 / 加载器。链接器 / 加载

器找到所需的例程，重新映射它，并更改间接跳转位置中的地址以指向该例程。然后跳转到这个例程。例程完成后，它将返回到初始调用点。此后，它都会间接跳转到该例程而不需额外的中间过程。

总之，DLL 需要额外的空间来存储动态链接所需的信息，但不要求复制或链接整个库。它们在第一次调用例程时会付出大量的开销，但此后只需一个间接跳转。请注意，从库返回不会产生额外的开销。微软的 Windows 广泛依赖动态链接库，如今 UNIX 系统上程序执行的默认设置也是使用动态链接库。

2.12.6 启动 Java 程序

上面的讨论主要描述了执行程序的传统模型，重点在于以特定指令系统体系结构甚至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 一样执行 Java 程序。然而，Java 是为了不同的目标而发明的，目标之一是能在任何计算机上安全运行，即使它可能会延长执行时间。

图 2-22 显示了 Java 典型的转换和执行步骤。Java 不是编译成目标计算机的汇编语言，而是首先编译成易于解释的指令：**Java 字节码**指令系统（参见 2.15 节）。该指令系统被设计得非常接近 Java 语言，因此编译步骤相对简单，实际上没有进行任何优化。与 C 编译器一样，Java 编译器检查数据类型并为每种类型生成正确的操作。Java 程序转化为这些字节码的二进制形式。

Java 字节码：为解释 Java 程序而设计的指令系统中的指令。

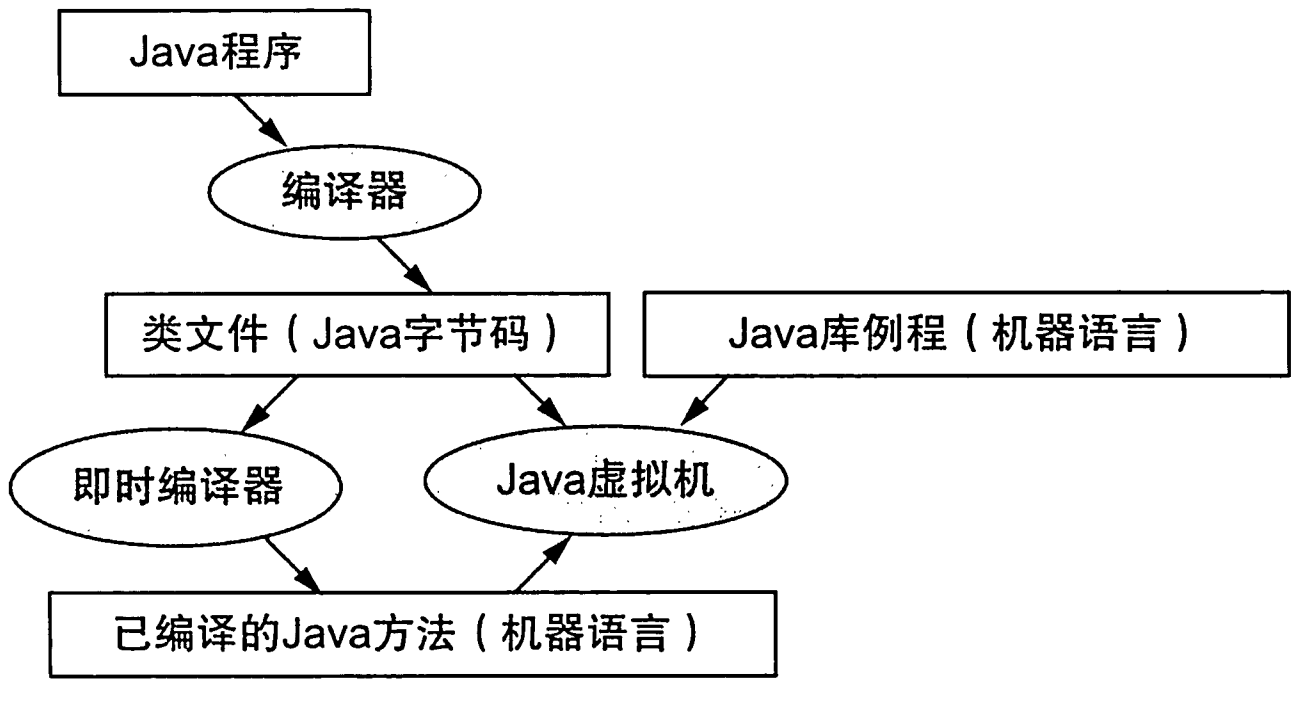


图 2-22 Java 的转换层次结构。Java 程序首先被编译成 Java 字节码的二进制版本，所有地址都由编译器定义。Java 程序现在可以在解释器上运行，称为 Java 虚拟机（JVM）。程序运行时，JVM 链接到 Java 库中所需的方法。为了获得更高的性能，JVM 可以调用 JIT 编译器，该编译器有选择地将方法编译为运行它的机器的本地机器语言

称为 **Java 虚拟机**（JVM）的软件解释器可以执行 Java 字节码。解释器是一个模拟指令系统体系结构的程序。例如，本书使用的 RISC-V 模拟器是一个解释器。由于转换非常简单，所以地址可以由编译器填写或在运行时被 JVM 发现，不需要单独的汇编步骤。

Java 虚拟机：解释 Java 字节码的程序。

解释的优点是可移植性。Java 虚拟机软件的可用性意味着大多数人可以在 Java 发布后不久编写和运行 Java 程序。如今，Java 虚拟机可以在数十亿台设备中找到，从手机到互联网浏览器。

解释的缺点是性能较低。20 世纪 80 年代和 90 年代令人难以置信的性能提升使得许多重要应用程序的解释成为可能，但与传统编译的 C 程序相比，10 倍的速度差距使 Java 对某些应用程序没有吸引力。

为了保持可移植性并提高执行速度，Java 发展的下一阶段目标是设计在程序运行时翻译的编译器。这样的**即时编译器**（JIT）通常会对正在运行的程序进行剖视，以找到“热点”方法所在的位置，然后将它们翻译成（运行虚拟机的）宿主机对应的指令。编译过的部分将在下次运行程序时保存，以便每次运行时速度更快。这种解释和编译的平衡随着时间的推移而发展，因此经常运行的 Java 程序几乎没有解释的开销。

即时编译器：对一类编译器的通用名称，该类编译器可以在运行时将已解释过的代码段翻译为宿主机上的机器语言。

随着计算机的速度变得越来越快，编译器也变得更为强大，并且随着研究人员发明出更好的动态编译 Java 的方法，Java 与 C 或 C ++ 之间的性能差距正在缩小。2.15 节将更深入地介绍 Java、Java 字节码、JVM 和 JIT 编译器。

自我检测　对于 Java 的设计者来说，解释器相对于翻译器的哪些优势最重要？

1. 解释器易于编写
2. 更准确的错误信息
3. 更少的目标代码
4. 机器独立性

2.13　以 C 排序程序为例的汇总整理

以片段形式展示汇编语言代码的一个危险是，读者将不知道完整的汇编语言程序是怎样的。在本节中，我们给出了两个 C 过程的 RISC-V 代码：一个用于交换（swap）数组元素，另一个用于对它们进行排序（sort）。

2.13.1　swap 过程

让我们从图 2-23 中交换过程的代码开始。此过程是交换内存中的两个位置。当手动把 C 程序翻译成汇编语言时，我们遵循以下步骤：

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

本节按这 3 个部分描述 swap 过程，最后将所有部分合并到一起。

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

图 2-23　交换内存中两个位置的 C 过程。本小节在排序示例中使用此过程

swap 的寄存器分配

如 2.8 节中所述，RISC-V 的参数传递默认使用寄存器 x10 到 x17。由于 swap 只有两个参数 v 和 k，因此可以在寄存器 x10 和 x11 中保存。唯一的一个变量是 temp，我们用寄存器 x5 来保存它，因为 swap 是一个叶过程（参见 2.8.2 节）。该寄存器分配对应于图 2-23 中 swap 过程第一部分的变量声明。

swap 过程体的代码

swap 剩余的 C 代码如下所示：

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

回想一下，RISC-V 的内存地址是字节寻址，因此双字实际上相差 8 个字节。因此，在将索引 k 与地址相加之前，需要将索引 k 乘以 8。忘记相邻的双字地址间相差 8 而不是 1 是汇编语言编程中常见的错误。因此，第一步是通过左移 3 位使 k 乘 8 以得到 v[k] 的地址：

```
slli    x6, x11, 3        // reg x6 = k * 8
add     x6, x10, x6       // reg x6 = v + (k * 8)
```

现在我们使用 x6 加载 v[k]，然后通过向 x6 加 8 来加载 v[k + 1]：

```
ld    x5, 0(x6)     // reg x5 (temp) = v[k]
ld    x7, 8(x6)     // reg x7 = v[k + 1]
                    // refers to next element of v
```

接下来，我们将 x5 和 x7 中的值存储到交换的地址：

```
sd    x7, 0(x6)     // v[k] = reg x7
sd    x5, 8(x6)     // v[k+1] = reg x5 (temp)
```

现在我们已经分配了寄存器并且翻译好了程序的代码。剩下的是保存 swap 程序中使用过的保留寄存器。因为在这个叶过程中没有使用保留寄存器，所以没有需要保存的寄存器。

完整的 swap 过程

我们现在已经得到完整的例程。剩下的就是添加过程标签和返回跳转。

```
swap:
  slli    x6, x11, 3   // reg x6 = k * 8
  add     x6, x10, x6  // reg x6 = v + (k * 8)
  ld      x5, 0(x6)    // reg x5 (temp) = v[k]
  ld      x7, 8(x6)    // reg x7 = v[k + 1]
  sd      x7, 0(x6)    // v[k] = reg x7
  sd      x5, 8(x6)    // v[k+1] = reg x5 (temp)
  jalr    x0, 0(x1)    // return to calling routine
```

2.13.2 sort 过程

为了确保读者能够理解汇编语言中编程的严谨性，我们将尝试第二个更长的示例。在这种示例中，我们将构建一个调用 swap 过程的例程。这个程序使用冒泡或交换排序对整数数组进行排序，这是最简单的排序之一，但不是最快的排序。图 2-24 显示了该程序的 C 语言版本。我们还是以几个步骤介绍此程序，并在最后将它们组合到一起。

```
void sort (long long int v[], size_t int n)
{
    size_t 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 2-24　对数组 v 执行排序的 C 过程

sort 的寄存器分配

过程 sort 的 v 和 n 两个参数保存在参数寄存器 x10 和 x11 中，我们将寄存器 x19 分配给 i，并将 x20 分配给 j。

sort 过程体的代码

过程体由两个嵌套的 for 循环和一个包含参数的 swap 调用组成。让我们从外而内展开代码。

第一个翻译步骤是第一个 for 循环：

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

回想一下，C 的 for 语句有三个部分：初始化、循环判断和循环增值。只需要一条指令就可以将 i 初始化为 0，这是 for 语句的第一部分：

```
li   x19,  0
```

（请记住，li 是汇编器为了方便汇编语言程序员而提供的伪指令；请参阅 2.12.2 节。）它只需要一条指令来递增 i，即 for 语句的最后一部分：

```
addi  x19, x19, 1  // i += 1
```

如果 i<n 非真，则应该退出循环，换句话说，如果 i ≥ n，则应该退出。此判断只需一条指令：

```
forltst: bge x19, x11, exit1  // go to exit1 if x19 ≥ x1 (i≥n)
```

循环的底部只是跳转回到循环判断处：

```
        j  forltst    // branch to test of outer loop
exit1:
```

那么第一个 for 循环的代码框架是

```
    li x19, 0                 // i = 0
forltst:
    bge x19, x11, exit1     // go to exit1 if x19 ≥ x1 (i≥n)
        ...
        (body of first for loop)
        ...
        addi x19, x19, 1    // i += 1
        j  forltst          // branch to test of outer loop
exit1:
```

瞧！（练习部分将探究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) {
```

该循环的初始化部分仍然是一条指令：

```
addi    x20, x19, -1     // j = i - 1
```

循环结束时 j 的自减也是一条指令：

```
addi    x20, x20, -1 j -= 1
```

循环判断有两个部分。如果任一条件为假，我们退出循环，因此如果第一个判断（j<0）为假则必须退出循环：

```
for2tst:
      blt x20, x0, exit2   // go to exit2 if x20 < 0 (j < 0)
```

该分支将跳过第二个条件判断。如果没有跳过，则 j ≥ 0。

如果第二个判断 v[j] > v[j+1] 非真，或者如果 v[j] ≤ v[j+1]，则退出。首先，我们通过将 j 乘以 8（因为我们需要一个字节地址）来创建地址并将其加到 v 的基址上：

```
slli      x5, x20, 3          // reg x5 = j * 8
add       x5, x10, x5         // reg x5 = v + (j * 8)
```

现在我们加载 v[j]：

```
ld        x6, 0(x5)           // reg x6 = v[j]
```

因为我们知道第二个元素是紧跟的一个双字，所以，将寄存器 x5 中的地址加 8，得到 v[j+1]：

```
ld        x7, 8(x5)           // reg x7 = v[j + 1]
```

我们判断 v[j] ≤ v[j+1] 以退出循环：

```
ble       x6, x7, exit2       // go to exit2 if x6 ≤ x7
```

循环的底部跳转回到内循环判断处：

```
j         for2tst             // branch to test of inner loop
```

将这些部分结合到一起，第二个 for 循环的代码框架是这样的：

```
            addi x20, x19, -1   // j = i - 1
for2tst:    blt x20, x0, exit2  // go to exit2 if x20 < 0 (j < 0)
            slli   x5, x20, 3     // reg x5 = j * 8
            add    x5, x10, x5    // reg x5 = v + (j * 8)
            ld     x6, 0(x5)      // reg x6 = v[j]
            ld     x7, 8(x5)      // reg x7 = v[j + 1]
            ble    x6, x7, exit2  // go to exit2 if x6 ≤ x7
                 . . .
                 (body of second for loop)
                 . . .
            addi x20, x20, -1     // j -= 1
            j      for2tst        // branch to test of inner loop
exit2:
```

sort 中的过程调用

下一步是翻译第二个 for 循环的循环体：

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

调用 swap 很容易：

```
jal x1, swap
```

sort 中的参数传递

当传递参数时问题就出现了，因为 sort 过程需要寄存器 x10 和 x11 中的值，但 swap 过程需要将其参数放在那些相同的寄存器中。一种解决方案是在过程较早的地方将 sort 中的参数复制到其他寄存器中，使得寄存器 x10 和 x11 在调用 swap 时可用。(这个复制比在栈上保存和恢复更快。) 我们首先在过程中将 x10 和 x11 复制到 x21 和 x22：

```
mv x21, x10          // copy parameter x10 into x21
mv x22, x11          // copy parameter x11 into x22
```