本书导读

在所有抽象的底部是计算机的五个经典部件：数据通路、控制器、存储器、输入和输出（见图 1-5）。这五个部件也是本书后面几章的框架：

- 数据通路：第 3、4、6 章和附录 B。
- 控制器：第 4、6 章和附录 B。
- 存储器：第 5 章。
- 输入：第 5 章和第 6 章。
- 输出：第 5 章和第 6 章。

如上所述，第 4 章介绍处理器如何开发隐式并行性，第 6 章介绍并行变革的核心——显式并行多核微处理器，附录 B 介绍高度并行的图形处理器芯片。第 5 章介绍如何层次化挖掘存储结构的访问局部性。第 2 章介绍指令系统——编译器和计算机之间的接口——并强调了编译器和编程语言在利用指令系统特性方面的作用。附录 A 提供了第 2 章指令系统的参考数据。第 3 章介绍计算机如何处理算术运算数据。附录 A 对逻辑设计进行了介绍。

1.12　历史视角和拓展阅读

本书的每一章都有“历史视角和拓展阅读”一节，可在配套网站上找到。我们可能通过一个系列的计算机来追踪某一思想的发展历程，或者描述一些历史上的重要项目，如果读者有兴趣进一步探究的话，我们还提供相关参考资料。

> 活跃的科学领域就像一个巨大的蚁丘；人们消失在互相对立的观点中，以光速传递着信息，将信息从一个地方传到另一个地方。
> *Lewis Thomas，《细胞生命的礼赞》中的“自然科学”，1974*

本章的“历史视角”为一些关键思想提供了相关历史背景知识，其目的是向读者介绍对技术进步做出贡献的重要历史人物及其事迹，并将其成就置于当时的历史背景中进行思考。温故知新，读者也许能更好地理解那些将影响未来计算技术的力量。配套网站中每个历史视角之后都会有“拓展阅读”的提示，这部分内容具体见配套网站中的“Further Reading”部分。1.12 节的剩余部分可在配套网站上在线查找。

1.13　练习

完成练习所需的相对时间标示在题号之后的方括号中。平均来说，在标级［10］的习题上所用的时间会是标级［5］的习题的 2 倍。做题前应先阅读的课本章节则标示在尖括号中。例如，<1.4> 表示应该阅读 1.4 节以帮助你完成该习题。

1.1　［2］<1.1>除了拥有数十亿用户的智能手机之外，请再列举和描述四种类型的计算机。

1.2　［5］<1.2>计算机体系结构中的八个伟大思想与其他领域的思想相似。请将计算机体系结构中的八个伟大思想——“面向摩尔定律的设计”“使用抽象简化设计”“加速经常性事件”“通过并行提高性能”“通过流水线提高性能”“通过预测提高性能”“存储器层次”“通过冗余提高可靠性”与其他领域的下列思想进行匹配：

a. 汽车制造中的组装生产线。

b. 吊桥缆索。

c. 采用风向信息的飞机和船舶导航系统。

d. 高楼中的高速电梯。

e. 图书馆的存阅处。

f. 通过增大 CMOS 晶体管的栅极面积来减少翻转时间。

g. 增加电磁飞机弹射器（不同于当前的蒸汽驱动模型，它采用电驱动），这可以通过新型反应堆技术增加的电能来实现。

h. 制造自动驾驶汽车，其控制系统是安装在汽车上的传感器系统，例如车道偏离检测系统和智能导航控制系统。

1.3 [2] <1.3> 请描述高级语言（例如 C）编写的程序转化为能够直接在计算机处理器上执行的表示的具体步骤。

1.4 [2] <1.4> 一个彩色显示器中的每个像素由三种基色（红、绿、蓝）构成，每种基色用 8 位表示，帧大小为 1280×1024。

a. 为了保存一帧图像最少需要多大的帧缓冲（以字节计算）？

b. 在 100Mbps 的网络上传输一帧图像最少需要多长时间？

1.5 [4] <1.6> 有 3 种不同的处理器 P1、P2 和 P3 执行同样的指令系统。P1 的时钟频率为 3GHz，CPI 为 1.5；P2 的时钟频率为 2.5GHz，CPI 为 1.0；P3 的时钟频率为 4GHz，CPI 为 2.2。

a. 以每秒执行的指令数为标准，哪个处理器性能最高？

b. 如果每个处理器执行一个程序都花费 10 秒时间，求它们的时钟周期数和指令数。

c. 我们试图把执行时间减少 30%，但这会引起 CPI 增大 20%。请问为达到时间减少 30% 的目标，时钟频率应达到多少？

1.6 [20] <1.6> 同一个指令系统体系结构有两种不同的实现方式。根据 CPI 的不同将指令分成四类（A、B、C 和 D）。P1 的时钟频率为 2.5GHz，CPI 分别为 1、2、3 和 3；P2 时钟频率为 3GHz，CPI 分别为 2、2、2 和 2。

给定一个程序，有 1.0×10^6 条动态指令，按如下比例分为 4 类：A，10%；B，20%；C，50%；D，20%。

a. 每种实现方式下的整体 CPI 是多少？

b. 计算两种情况下的时钟周期总数。

1.7 [15] <1.6> 编译器对应用程序性能有极深的影响。假定对于一个程序，如果采用编译器 A，则动态指令数为 1.0×10^9，执行时间为 1.1s；如果采用编译器 B，则动态指令数为 1.2×10^9，执行时间为 1.5s。

a. 在给定处理器时钟周期长度为 1ns 时，求每个程序的平均 CPI。

b. 假定被编译的程序分别在两个不同的处理器上运行。如果这两个处理器的执行时间相同，求运行编译器 A 生成之代码的处理器时钟比运行编译器 B 生成之代码的处理器时钟快多少。

c. 假设开发了一种新的编译器，只需 6.0×10^8 条指令，程序平均 CPI 为 1.1。求这种新的编译器在原处理器环境下相对于原编译器 A 和 B 的加速比。

1.8 2004 年发布的 Pentium 4 Prescott 处理器时钟频率为 3.6GHz，工作电压为 1.25V。假定平均情况下静态功耗为 10W，动态功耗为 90W。

2012 年发布的 Core i5 Ivy Bridge 时钟频率为 3.4GHz，工作电压为 0.9V。假定平均情况下静态功耗为 30W，动态功耗为 40W。

1.8.1 [5] <1.7> 分别求出每个处理器的平均电容负载。

1.8.2 [5] <1.7> 对于每种技术，求出静态功耗占总耗散功耗的比例和静态功耗相对于动态功耗的比率。

1.8.3 [15] < 1.7 > 如果要将整体耗散功耗降低 10%，请计算出在保持漏电流不变的情况下电压要降低多少。注意：功率定义为电压与电流的乘积。

1.9 在某处理器中，假定算术指令、load/store 指令和分支指令的 CPI 分别是 1、12 和 5。同时假定某程序在单个处理器核上运行时需要执行 2.56×10^9 条算术指令、1.28×10^9 条 load/store 指令和 2.56×10^8 条分支指令，并假定处理器的时钟频率为 2GHz。

现假定程序并行运行在多核上，分配到每个处理器核上运行的算术指令和 load/store 指令数目为单核情况下相应指令数目除以 $0.7 \times p$（p 为处理器核数），而每个处理器的分支指令的数量保持不变。

1.9.1 [5] < 1.7 > 求出当该程序分别运行在 1、2、4 和 8 个处理器核上的执行时间，并求出其他情况下相对于单核处理器的加速比。

1.9.2 [10] < 1.6, 1.8 > 如果算术指令的 CPI 加倍，对分别运行在 1、2、4 和 8 个处理器核上的执行时间有何影响？

1.9.3 [10] < 1.6, 1.8 > 如果要使单核处理器的性能与四核处理器相当，单处理器中 load/store 指令的 CPI 应该降低多少？此处假定四核处理器的 CPI 保持原数值不变。

1.10 假定一个直径 15cm 的晶圆的成本是 12，包含 84 枚晶片，其缺陷参数为 0.020 个 /cm^2。而一个直径 20cm 的晶圆的成本是 15，包含 100 枚晶片，其缺陷参数为 0.031 个 /cm^2。

1.10.1 [10] < 1.5 > 分别求出每种晶圆的工艺良率。

1.10.2 [5] < 1.5 > 分别求出每种晶片的价格。

1.10.3 [5] < 1.5 > 如每晶圆的晶片数增加 10%，每单位面积的缺陷数增加 15%，求晶片面积和工艺良率。

1.10.4 [5] < 1.5 > 假设随着电子器件制造技术的进步，工艺良率从 0.92 上升到 0.95。给定晶片面积为 200mm^2，求每一种技术下单位面积的缺陷数。

1.11 SPEC CPU2006 的 bzip2 基准程序在 AMD Barcelona 处理器上执行的总指令数为 2.389×10^{12}，执行时间为 750 秒，该程序的参考执行时间为 9650 秒。

1.11.1 [5] < 1.6,1.9 > 如果时钟周期长度为 0.333ns，求 CPI 值。

1.11.2 [5] < 1.9 > 求该程序的 SPEC 分值。

1.11.3 [5] < 1.6, 1.9 > 如果基准程序的指令数增加 10%，CPI 不变，则 CPU 时间增加多少？

1.11.4 [5] < 1.6, 1.9 > 如果基准程序的指令数增加 10%，CPI 增加 5%，则 CPU 时间增加多少？

1.11.5 [5] <1.6, 1.9 > 根据上题中指令数和 CPI 的变化，求 SPEC 分值的变化。

1.11.6 [10] < 1.6 > 假设正在开发一款新的 AMD Barcelona 处理器，其工作频率为 4GHz，在其指令系统中增加了一些新的指令，从而使程序中的指令数目减少了 15%，程序的执行时间降到了 700 秒，新的 SPEC 分值为 13.7，求新的 CPI。

1.11.7 [10] < 1.6 > 当时钟频率由 3GHz 上升到 4GHz 时，上一题算出的 CPI 比 1.11.1 的高。请确定 CPI 的升高是否与频率升高相同。如果不同，为什么？

1.11.8 [5] < 1.6 >CPU 时间减少了多少？

1.11.9 [10] < 1.6 > 对第二个基准程序 libquantum，假定执行时间为 960ns,CPI 为 1.61，时钟频率为 3GHz。在时钟频率为 4GHz 时，在不影响 CPI 的前提下执行时间减少 10%，求指令总数。

1.11.10 [10] < 1.6 > 在指令总数和 CPI 保持不变的前提下，如果要将 CPU 时间进一步减少 10%，求时钟频率。

1.11.11 [10] < 1.6 > 在指令总数保持不变的前提下，如果要将 CPI 降低 15%,CPU 时间减少 20%，求时钟频率。

1.12 在 1.10 节中提到使用性能公式的子集来作为性能评价指标的陷阱。下面的习题将对其进行说明。考虑下面两种处理器。P1 的时钟频率为 4GHz，平均 CPI 为 0.9，需要执行 5.0×10^9 条指令；P2 的时钟频率为 3GHz，平均 CPI 为 0.75，需要执行 1.0×10^9 条指令。

1.12.1 [5] <1.6, 1.10> 一个常见的错误是，认为时钟频率最高的计算机具有最高的性能。这种说法正确吗？请用 P1 和 P2 来验证这一说法是否正确。

1.12.2 [10] <1.6, 1.10> 另一个错误是，认为执行指令最多的处理器需要更多的 CPU 时间。考虑 P1 执行 1.0×10^9 条指令序列所需的时间，假定 P1 和 P2 的 CPI 不变，计算一下 P2 用同样的时间可以执行多少条指令。

1.12.3 [10] <1.6, 1.10> 一个常见的错误是用 MIPS（每秒百万条指令数）来比较两台不同的处理器的性能，并认为 MIPS 数值大的处理器具有最高的性能。这种说法正确吗？请用 P1 和 P2 验证这一说法是否正确。

1.12.4 [10] <1.10> 另一个常见的性能标志是 MFLOPS（每秒百万条浮点指令），其定义为

$$\text{MFLOPS} = \frac{\text{浮点操作的数目}}{\text{执行时间} \times 10^6}$$

但此指标与 MIPS 有同样的问题。假定 P1 和 P2 上执行的指令有 40% 的浮点指令，求出各处理器的 MFLOPS 数值。

1.13 在 1.10 节中提到的另一个易犯的错误是希望通过只改进计算机的一个方面来改进计算机的总体性能。假如一台计算机上运行一个程序需要 250 秒，其中 70 秒用于执行浮点指令，85 秒用于执行 L/S 指令，40 秒用于执行分支指令。

1.13.1 [5] <1.10> 如果浮点操作的时间减少 20%，总时间将减少多少？

1.13.2 [5] <1.10> 如果将总时间减少 20%，整型操作时间应减少多少？

1.13.3 [5] <1.10> 如果只减少分支指令时间，总时间能否减少 20%？

1.14 假定一个程序需要执行 50×10^6 条浮点指令、110×10^6 条整型指令、80×10^6 条 L/S 指令和 16×10^6 条分支指令。每种类型指令的 CPI 分别是 1、1、4 和 2。假定处理器的时钟频率为 2GHz。

1.14.1 [10] <1.10> 如果我们要将程序运行速度提高至原来的 2 倍，浮点指令的 CPI 应如何改进？

1.14.2 [10] <1.10> 如果我们要将程序运行速度提高至原来的 2 倍，L/S 指令的 CPI 应如何改进？

1.14.3 [5] <1.10> 如果整数和浮点指令的 CPI 减少 40%，L/S 和分支指令的 CPI 减少 30%，程序的执行时间能改进多少？

1.15 [5] <1.8> 当程序被调整到多核处理器系统的多个处理器上运行时，在每个处理器上的执行时间可分成计算时间、由于互锁临界区和 / 或处理器之间传输数据的通信带来的时间开销。

假定一个程序在单处理器上执行需要的时间 t 为 100 秒。当它在 p 个处理器上运行时，每个处理器需要 t/p 秒的计算时间，以及 4 秒的额外开销，且此开销与处理器数量无关。在处理器数目分别为 2、4、8、16、32、64 和 128 时，计算每个处理器的执行时间。在每种情况下，列出相对于单处理器的加速比和实际加速比与理想加速比的比值（理想加速比是指没有开销情况下的加速比）。

自我检测答案

1.1 节　问题讨论：可以有多种答案。

1.4 节　DRAM 存储器：具有易失性，访问时间短（大约 50 ～ 70ns），每 GB 的价格为 5 ～ 10

美元。磁盘存储器：具有非易失性，访问时间比DRAM慢100 000～400 000倍，每GB的价格比DRAM便宜100倍。闪存：具有非易失性，访问时间比DRAM慢100～1000倍，每GB的价格比DRAM便宜7～10倍。

1.5节　1、3、4是正确答案，答案5一般可认为正确，因为产量高时能促使额外投资去缩小晶片面积，例如减小10%从经济上来看就是很好的决策了，但这往往不太现实。

1.6节　1.a. 两者都改进；b. 延迟；c. 都不改进；2.7秒。

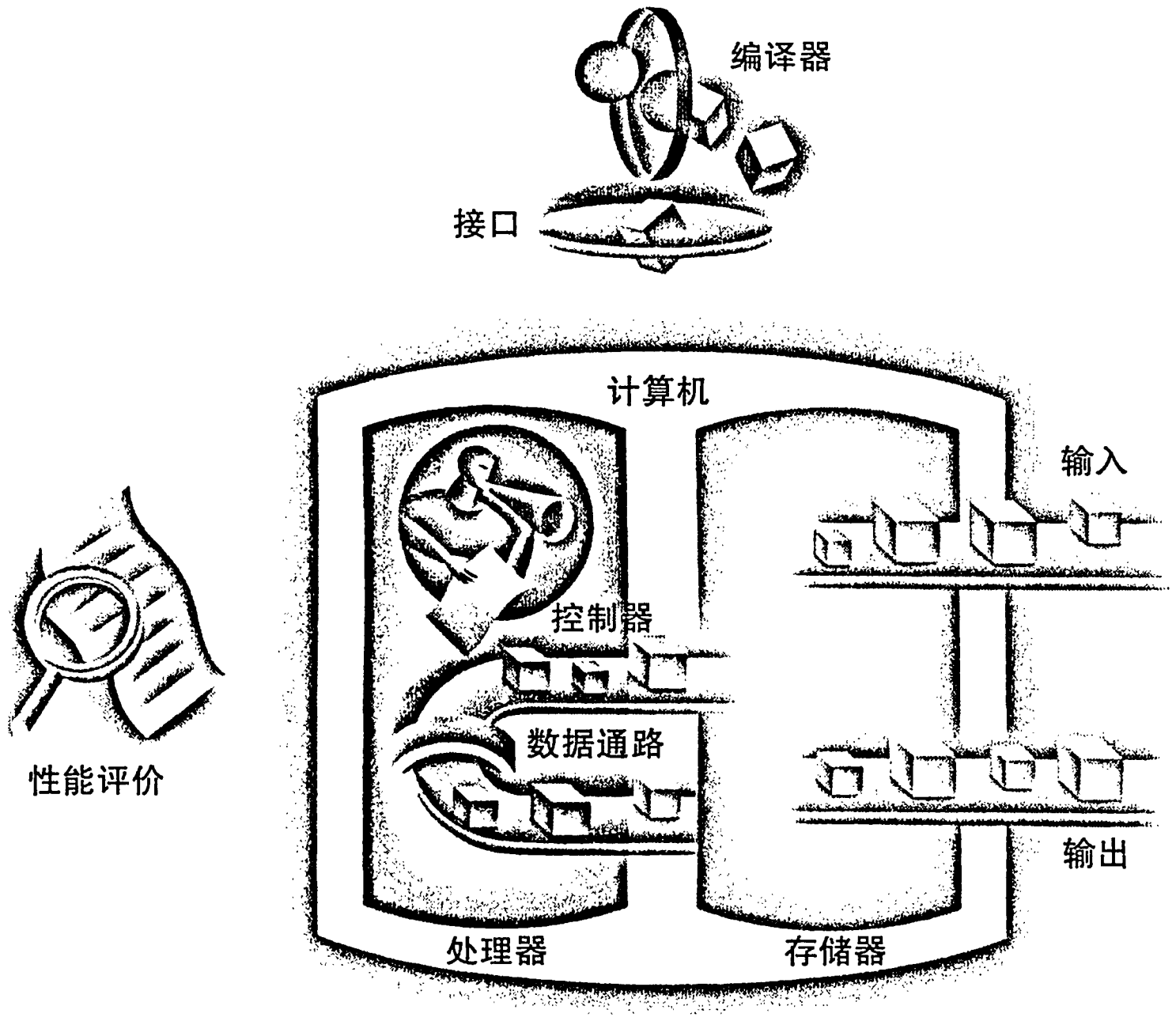
1.6节　b

1.10节　a. 计算机A有较高的MIPS值；b. 计算机B更快。

第 2 章

Computer Organization and Design: The Hardware/Software Interface, RISC-V Edition

指令：计算机的语言



计算机的五个经典部件

2.1 引言

要控制计算机硬件，就必须用它的语言。计算机语言中的单词称为指令，其词汇表称为**指令系统**（instruction set）。在本章中，你将看到一个真实计算机的指令系统，有人为书写和计算机可读两种形式。我们以自上而下的方式介绍指令。从一种看似受限的编程语言符号开始，逐步完善，直到成为计算机的实际语言。第 3 章继续这个向下的过程，揭示计算硬件和浮点数的表示。

我同上帝说西班牙语，和女人说意大利语，跟男人说法语，对我的马说德语。
查理五世，神圣罗马帝国皇帝（1500—1558）

你可能认为计算机语言和人类语言一样种类繁多，但实际上计算机语言都十分类似，更像是区域方言而非各自独立的语言。因此，一旦学会了一种，再学习其他语言就很容易了。

指令系统：被一个给定体系结构所理解的命令词汇表。

本书所选指令系统为 RISC-V，2010 年初由加州大学伯克利分校开发。

为了演示学会其他指令系统有多么容易，我们还将快速浏览另外两个流行的指令系统。

1. MIPS 是设计于 20 世纪 80 年代的指令系统典范。在许多方面，RISC-V 都遵循类似的设计。

2. Intel x86 起源于 20 世纪 70 年代，现在仍然支持 PC 以及后 PC 时代的云端。

指令系统的相似性是因为所有计算机都是基于相似基本原理的硬件技术构建的，且因为有些基本操作是所有计算机都必须提供的。此外，计算机设计人员都有一个共同目标：找到一种让构建硬件和编译器容易，同时最大化性能且最小化成本和资源的语言。这是个历史悠久的目标。下述引文写于计算机可购买前的1947年，但在今天同样适用：

> 通过形式逻辑方法很容易看出，存在某些在理论上足以控制和执行任意操作序列的[指令系统]……从现有观点来看，选择一个[指令系统]的真正决定性考虑因素是更具实用性：[指令系统]所要求的硬件简单性，及其应用于实际重要问题的清晰度以及处理这些问题的速度。
>
> Burks、Goldstine 和 von Neumann，1947

对于今天的计算机而言，“硬件简单性”与20世纪50年代一样值得考虑。本章的目标便是讲述一个遵循此想法的指令系统，分别展示它在硬件中如何表示，及其（更初级的语言）与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节展示了对于像Java这样的面向对象的语言，上述部分会有怎样的变化。

通过学习如何表示指令，还能发现计算的秘密：**存储程序概念**。此外，可以通过使用计算机语言编写程序，并在本书附带的模拟器上运行以练习这门“外语”技能。我们还将看到编程语言和编译器优化对性能的影响。在本章结束时将简要介绍指令系统的历史演变以及其他计算机语言。

存储程序概念：指令与多种类型的数据不加区别地存储在存储器中并因此易于更改，因此产生了存储程序计算机。

我们将逐步介绍第一个指令系统，同时给出计算机结构的基本原理。这个自顶向下、循序渐进的教程将组件与解释相结合，使计算机语言更加易于接受。图2-1给出了本章所包含指令系统的预览。

RISC-V 操作数

名字	示例	注解
32个寄存器	x0～x31	快速定位数据。在RISC-V中，只对在寄存器中的数据执行算术运算
2^{61}个存储字	Memory[0], Memory[8], ..., Memory[18 446 744 073 709 551 608]	只能被数据传输指令访问。RISC-V使用字节寻址，因此顺序双字访问相差8。存储器保存数据结构、数组和换出的寄存器的内容

RISC-V 汇编语言

类别	指令	示例	含义	注解
算术运算	加	add x5, x6, x7	x5 = x6 + x7	三寄存器操作数；加
	减	sub x5, x6, x7	x5 = x6 - x7	三寄存器操作数；减
	立即数加	addi x5, x6, 20	x5 = x6 + 20	用于加常数
数据传输	取双字	ld x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取双字到寄存器
	存双字	sd x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存双字到存储器
	取字	lw x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取字到寄存器
	取字（无符号数）	lwu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号字到寄存器
	存字	sw x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字到存储器
	取半字	lh x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取半字到寄存器
	取半字（无符号数）	lhu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号半字到寄存器

图2-1　本章中出现的RISC-V汇编语言。此信息列在本书RISC-V参考数据卡[⊖]的第1列中

⊖　见本书封面和封底的背面。——编辑注

RISC-V 汇编语言

类别	指令	示例	含义	注释
数据传输	存半字	sh x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存半字到存储器
	取字节	lb x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取字节到寄存器
	取字节（无符号数）	lbu x5, 40(x6)	x5 = Memory[x6 + 40]	从存储器取无符号字节到寄存器
	存字节	sb x5, 40(x6)	Memory[x6 + 40] = x5	从寄存器存字节到存储器
	取保留字	lr.d x5, (x6)	x5 = Memory[x6]	取；原子交换的前半部分
	存条件字	sc.d x7, x5, (x6)	Memory[x6] = x5; x7 = 0/1	存；原子交换的后半部分
	取立即数高位	lui x5, 0x12345	x5 = 0x12345000	取左移12位后的20位立即数
逻辑运算	与	and x5, x6, x7	x5 = x6 & x7	三寄存器操作数；按位与
	或	or x5, x6, x8	x5 = x6 | x8	三寄存器操作数；按位或
	异或	xor x5, x6, x9	x5 = x6 ^ x9	三寄存器操作数；按位异或
	与立即数	andi x5, x6, 20	x5 = x6 & 20	寄存器与常数按位与
	或立即数	ori x5, x6, 20	x5 = x6 | 20	寄存器与常数按位或
	异或立即数	xori x5, x6, 20	x5 = x6 ^ 20	寄存器与常数按位异或
移位操作	逻辑左移	sll x5, x6, x7	x5 = x6 << x7	按寄存器给定位数左移
	逻辑右移	srl x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数右移
	算术右移	sra x5, x6, x7	x5 = x6 >> x7	按寄存器给定位数算术右移
	逻辑左移立即数	slli x5, x6, 3	x5 = x6 << 3	根据立即数给定位数左移
	逻辑右移立即数	srli x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数右移
	算术右移立即数	srai x5, x6, 3	x5 = x6 >> 3	根据立即数给定位数算术右移
条件分支	相等即跳转	beq x5, x6, 100	if (x5 == x6) go to PC+100	若寄存器数值相等则跳转到PC相对地址
	不等即跳转	bne x5, x6, 100	if (x5 != x6) go to PC+100	若寄存器数值不等则跳转到PC相对地址
	小于即跳转	blt x5, x6, 100	if (x5 < x6) go to PC+100	若寄存器数值比较结果小于则跳转到PC相对地址
	大于等于即跳转	bge x5, x6, 100	if (x5 >= x6) go to PC+100	若寄存器数值比较结果大于或等于则跳转到PC相对地址
	小于即跳转（无符号）	bltu x5, x6, 100	if (x5 < x6) go to PC+100	若寄存器数值比较结果小于则跳转到PC相对地址（无符号）
	大于等于即跳转（无符号）	bgeu x5, x6, 100	if (x5 >= x6) go to PC+100	若寄存器数值比较结果大于或等于则跳转到PC相对地址（无符号）
无条件跳转	跳转-链接	jal x1, 100	x1 = PC+4; go to PC+100	用于PC相关的过程调用
	跳转-链接（寄存器地址）	jalr x1, 100(x5)	x1 = PC+4; go to x5+100	用于过程返回；非直接调用

图 2-1 （续）

2.2 计算机硬件的操作

每台计算机都必须能够实现算术运算。RISC-V 汇编语言的符号

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

指示计算机将两个变量 b 和 c 相加并将其总和放入 a 中。

> 毫无疑问，必须有实现基本算术运算的指令。
> *Burk、Goldstine 和 von Neumann，1947*

这种符号表示是固定的，其中每个 RISC-V 算术指令只执行一个操作，并且必须总是只有三个变量。例如，假设要将四个变量 b、c、d 和 e 的和放入变量 a 中。(本节不深究“变量”的概念，下一节将详细解释。)

以下指令序列将四个变量相加：

```
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
```

因此，需要三条指令来完成这四个变量相加。

上面每行双斜线（//）的右边是给读者的注释，而计算机会将其忽略。请注意，与其他编程语言不同，该语言每行最多只能包含一条指令。与 C 语言的另一个区别是注释总是在一行的末尾终止。

类似于加法的操作一般有三个操作数：两个被加到一起的数和一个放置总和的位置。要求每条指令恰好有三个操作数，不多也不少，符合硬件简单的设计原则：操作数数量可变的硬件比固定数量的硬件更复杂。这种情况说明了硬件设计三条基本原则之一：

设计原则 1：简单源于规整。

现使用以下两个示例展示用高级编程语言编写的程序和用初级符号表示的程序间的关系。

例题 | 将两条 C 赋值语句编译成 RISC-V

这段 C 程序代码包含五个变量 a、b、c、d 和 e。由于 Java 是从 C 发展而来的，所以本例以及接下来若干示例都适用于这两种高级编程语言：

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

编译器将 C 语言转换成 RISC-V 汇编指令。写出编译器生成的 RISC-V 代码。

答案 | RISC-V 指令对两个源操作数进行操作，并将结果放入一个目标操作数。因此，上面的两条简单语句直接编译成以下两条 RISC-V 汇编指令。

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

例题 | 将一条复杂的 C 赋值语句编译成 RISC-V

有一条包含五个变量 f、g、h、i 和 j 的复杂语句：

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

C 编译器可能产生什么样的 RISC-V 汇编指令？

答案 | 编译器必须将该语句分解为多条汇编指令，因为每条 RISC-V 指令只执行一个操作。第一条 RISC-V 指令计算 g 与 h 之和。因为必须将结果放于某处，所以编译器会创建一个名为 t0 的临时变量：

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

虽然下一个操作是减法，但是需要在做减法之前先计算 i 与 j 之和。因此，第二条指令将 i 和 j 的和放入由编译器创建的另一个名为 t1 的临时变量中：

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

最后，减法指令从第一条指令求得的和中减去第二条指令求得的和，并将差值放入变量 f 中，完成编译：

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

详细阐述 为了提高可移植性，Java 最初被设计为依赖于软件的解释器。这个解释器的指令系统称为 Java 字节码（bytecode，见 2.15 节），它与 RISC-V 指令系统有很大不同。为了使性能接近于等效的 C 程序，现在的 Java 系统通常会将 Java 字节码编译为像 RISC-V 这样的机器指令。由于完成这种编译通常比 C 程序晚得多，因此这样的 Java 编译器通常称为即时（Just In Time，JIT）编译器。2.12 节展示了在程序启动过程中，相对于 C 编译器，JIT 是如何延后使用的，2.13 节展示了 Java 程序编译和解释的性能比较。

自我检测 对于一个给定函数，哪种编程语言可能需要大量代码？将以下三种表示语言进行排序。

1. Java
2. C
3. RISC-V 汇编语言

2.3 计算机硬件的操作数

与高级语言程序不同，算术指令的操作数会受到限制；它们必须取自寄存器，而寄存器数量有限并内建于硬件的特殊位置。寄存器是硬件设计中的基本元素，当计算机设计完成后，对程序员也可见，因此可以将寄存器视为计算机构建的“砖块”。在 RISC-V 体系结构中，寄存器的大小为 64 位；成组的 64 位频繁出现，因此它们在 RISC-V 体系结构中被命名为**双字**。（另一个常见大小是成组的 32 位，在 RISC-V 体系结构中称为**字**。）

双字：计算机中一种访问基本单位，通常是64位一组；对应于RISC-V体系结构中寄存器的大小。

字：计算机中另一种访问基本单位，通常是32位一组。

程序语言的变量和寄存器之间的一个主要区别是寄存器数量有限，在当前 RISC-V 等计算机上通常为 32 个（关于寄存器数量的演变历史见 2.21 节）。因此，我们继续自顶向下逐步引入 RISC-V 语言的符号表示，在本节中，我们增加了限制，即 RISC-V 算术指令的三个操作数必须从 32 个 64 位寄存器中选择。

寄存器个数限制为 32 个的原因可以在我们硬件设计三条基本设计原则的第二条中找到。

设计原则 2：更少则更快。

简单来讲，数量过多的寄存器可能会增加时钟周期，因为电信号传输的距离越远，所花费的时间就越长。

诸如“更少则更快”的设计原则不是绝对的，31 个寄存器也许并不比 32 个更快。然而，这种发现背后的真相也需要计算机设计人员认真对待。在这种情况下，设计人员必须在程序对更多寄存器的渴求和设计人员对缩短时钟周期的期望之间取得平衡。另一个不使用超过 32 个寄存器的原因是指令格式的位数限制，如 2.5 节所介绍的。

第 4 章展示了寄存器在硬件构造中扮演的核心角色；正如将在该章中看到的，有效使用寄存器对于提高程序性能至关重要。

尽管我们可以简单地使用寄存器编号 0 到 31 来编写指令，但是 RISC-V 约定在“x”后面跟一个寄存器编号来表示，有一些例外的寄存器名称我们将稍后介绍。

例题 | 使用寄存器编译 C 赋值语句

编译器的工作是将程序变量与寄存器相关联。以我们前面例子中的赋值语句为例：

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

变量 f、g、h、i 和 j 分别分配给寄存器 x19、x20、x21、x22 和 x23。编译后的 RISC-V 代码是什么？

答案 | 编译后的程序与之前的例子非常相似，只是我们用上面提到的寄存器名称替换了变量，并将两个临时变量用两个临时寄存器 x5 和 x6 代替：

```
add x5, x20, x21  // register x5 contains g + h
add x6, x22, x23  // register x6 contains i + j
sub x19, x5, x6   // f gets x5 - x6, which is (g + h) - (i + j)
```

2.3.1 存储器操作数

程序语言中，有如同这些例子一样包含单个数据元素的简单变量，也有更复杂的数据结构——数组和结构体。这些复杂数据结构可以包含比计算机中寄存器数量更多的数据元素。计算机如何表示和访问这样庞大的数据结构呢？

回顾一下计算机的五个组成部分。处理器只能在寄存器中保存少量数据，但是计算机内存可以存储数十亿数据元素。因此，数据结构（数组和结构体）可以保存在内存中。

如上所述，RISC-V 指令中的算术运算只作用于寄存器，因此，RISC-V 必须包含在内存和寄存器之间传输数据的指令。这些指令称为**数据传输指令**。要访问内存中的字或双字，指令必须提供内存**地址**。内存只是一个大型一维数组，其地址作为该数组的下标，从 0 开始。例如，在图 2-2 中，第三个数据元素的地址是 2，内存第 2 号单元存放的数据是 10。

数据传输指令：在内存和寄存器之间传送数据的命令。

地址：用于描述内存数组中特定数据元素位置的值。

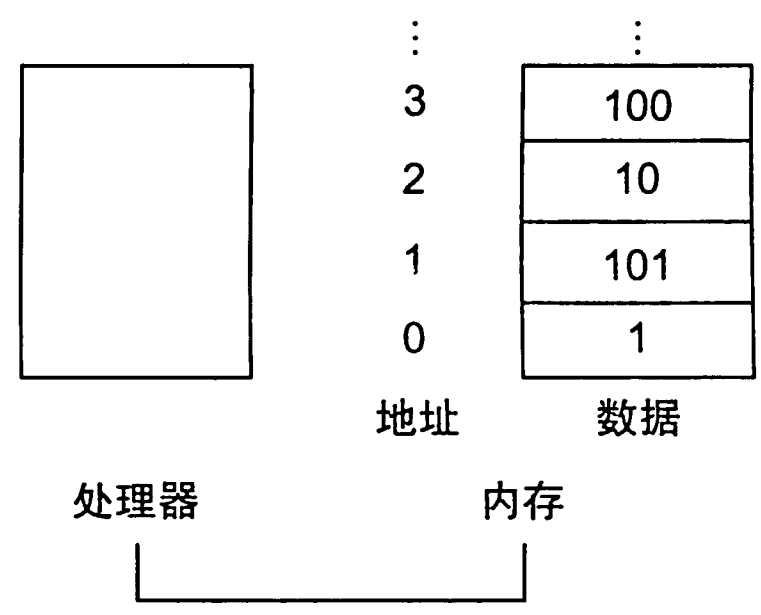


图 2-2　内存地址和地址中的内容。如果这些元素是双字，这些地址将是不正确的，因为 RISC-V 实际上使用字节寻址，每个双字代表 8 个字节。图 2-3 展示了顺序双字编址的正确内存寻址

将数据从内存复制到寄存器的数据传输指令通常称为载入指令（load）。载入指令的格式是操作名称后面紧跟数据待取的寄存器，然后是寄存器和用于访问内存的常量[⊖]。指令的常量部分和第二个寄存器中的内容[⊖]相加组成内存地址。实际的 RISC-V 指令名称是 ld，表示取双字。

例题｜当操作数在内存中时，编译 C 赋值语句

假设 A 是一个由 100 个双字组成的数组，并且编译器和之前一样将寄存器 x20 和 x21 分别分配给变量 g 和 h。我们还假设数组的起始地址或基址存放在寄存器 x22 中。编译这个 C 赋值语句：

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

⊖ 即偏移量。——译者注

⊖ 即基址。——译者注

答案 | 虽然在这个赋值语句中只有一个操作，但其中一个操作数在内存中，所以我们必须先将 A[8] 传送到一个寄存器。该数组元素的地址是数组 A 的基址（在寄存器 x22 中）加上元素序号 8 的和。数据应该放在一个临时寄存器中以便下一条指令使用。根据图 2-2，第一条编译后的指令是：

```
ld   x9, 8(x22) // Temporary reg x9 gets A[8]
```

（之后我们将对这条指令做轻微的调整，但现在使用简化的版本。）下一条指令可以对 x9 操作，因为 A[8] 已存放在寄存器 x9 中。该指令必须将 h（存放在 x21 中）和 A[8]（存放在 x9 中）相加，并将该和放入与 g 相对应的寄存器 x20 中：

```
add  x20, x21, x9 // g = h + A[8]
```

存放基址的寄存器（x22）被称为基址寄存器，而数据传输指令中的常数 8 称为偏移量。■

硬件/软件接口 除了将变量与寄存器相对应以外，编译器还将像数组和结构体这样的数据结构分配到内存中的相应位置。编译器可以将正确的起始地址放入数据传输指令中。

由于 8 位字节在许多程序中非常有用，几乎所有的体系结构都是按单个字节寻址的。因此，双字的地址与双字内的 8 个字节之一的地址是相匹配的，并且连续双字的地址相差 8。例如，图 2-3 显示了图 2-2 中双字的实际 RISC-V 地址，第三个双字的字节地址是 16。

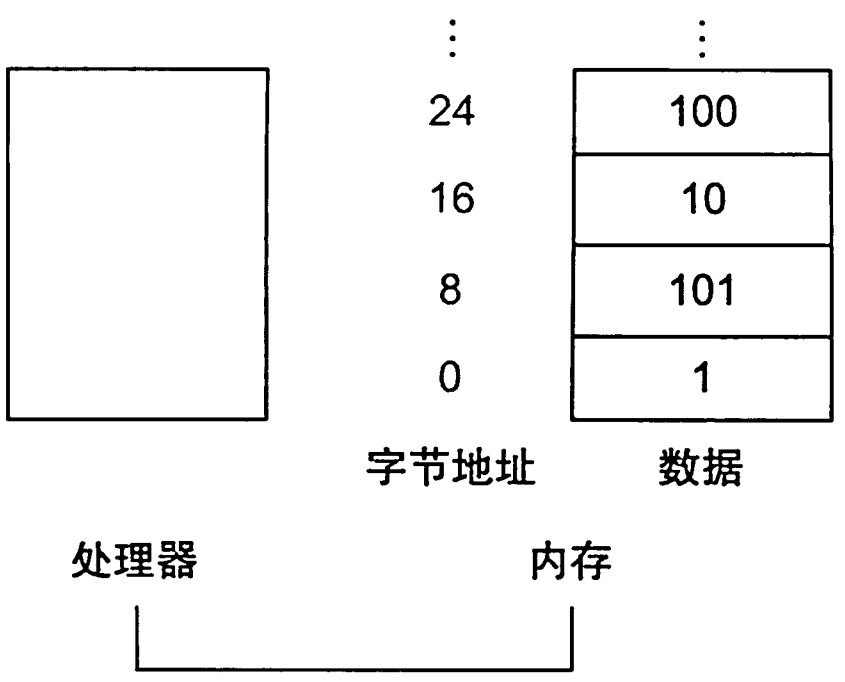


图 2-3 实际的 RISC-V 内存地址和这些内存中双字的内容。为了与图 2-2 对照，改变了的地址用灰色标出。由于 RISC-V 按字节寻址，因此双字地址是 8 的倍数：双字包含 8 个字节

计算机分为两种，一种使用最左边或“大端”字节的地址作为双字地址，另一种使用最右端或“小端”字节的地址作为双字地址。RISC-V 属于后者，称为小端编址。由于仅在以双字形式和八个单独字节访问相同数据时，字节顺序才会有影响，因此大多数情况下不需要关心“大小端”。

字节寻址也会影响数组下标。为了在上面的代码中获得正确的字节地址，加到基址寄存器 x22 的偏移量必须是 8×8 或 64，以便取地址将选择 A[8] 而不是 A[8/8]。（参见 2.19 节对相关陷阱的介绍。）

与载入指令相反的指令通常被称为存储指令（store），它从寄存器复制数据到内存。存储指令的格式类似于载入指令的格式：操作名称，接着是要写回内存的寄存器，然后是基址寄存器，最后是选择数组元素的偏移量。同样，RISC-V 地址是由常数和基址寄存器内容共同决定的。实际上的 RISC-V 指令名称是 sd，表示存储双字。

详细阐述 在许多体系结构中，字的起始地址必须是4的倍数，双字的起始地址必须是8的倍数。该要求称为对齐限制。（第4章将指出为什么对齐会使数据传输更快。）RISC-V和Intel x86没有对齐限制，但MIPS确实有此限制。

对齐限制：数据在内存中要与自然边界对齐的要求。

硬件/软件接口 由于加载和存储指令中的地址是二进制数，我们可以看到为什么作为主存的DRAM以二进制而不是十进制表示容量大小。也就是说，是以gibibyte（2^{30}）或tebibyte（2^{40}）表示，而不是gigabyte（10^9）或terabyte（10^{12}）；见图1-1。

例题｜使用load和store编译生成指令

假设变量h存放在寄存器x21中，数组A的基址存放在寄存器x22中。C赋值语句的RISC-V汇编代码是什么？

```
A[12] = h + A[8];
```

答案｜ 虽然在C语句中只有一个操作，但是现在在内存中有两个操作数，所以我们需要更多的RISC-V指令。前两个指令与前一个例子相同，只是这次我们在加载指令中使用了字节寻址正确的偏移量来选择A[8]，并且加法指令将总和放在寄存器x9中：

```
ld    x9, 64(x22)    // Temporary reg x9 gets A[8]
add   x9, x21, x9    // Temporary reg x9 gets h + A[8]
```

最后的指令使用96（8×12）作为偏移量，寄存器x22作为基址寄存器，将总和存储到A[12]中。

```
sd    x9, 96(x22)    // Stores h + A[8] back into A[12]
```

加载双字和存储双字是在RISC-V体系结构中存储器和寄存器之间传输双字的指令。某些品牌的计算机使用其他的载入和存储指令来传输数据。采用这种替代方案的一种体系结构是2.17节中描述的Intel x86。

硬件/软件接口 许多程序有着比计算机中寄存器数量更多的变量。所以，编译器会尽量把最常用的变量存放在寄存器中，剩下的存放在内存中，使用load和store在寄存器和内存之间传输变量。将不常用的变量（或稍后才使用的变量）存放到内存的过程称为寄存器换出。

关于大小和速度的硬件设计原则表明内存一定比寄存器慢，因为寄存器更少。事实确实如此，如果数据在寄存器而不是内存中，数据访问速度会更快。

而且，数据在寄存器中更有用。RISC-V算术指令可以读取两个寄存器，对它们进行操作并写入结果。RISC-V数据传输指令只读取一个操作数或写入一个操作数，并不对其进行操作。

因此，与内存相比，寄存器的访问时间更短，吞吐率更高。这使得寄存器中的数据访问速度更快，使用更简单。与访问内存相比，访问寄存器所需的能耗也少得多。要获得最高的性能并节约能耗，指令系统体系结构必须有足够多的寄存器，并且编译器必须有效使用寄存器。

详细阐述 让我们全面看待寄存器相对于内存的能耗和性能。假设访问一个64位数据，和2015年的DRAM相比，寄存器大约快200倍（50纳秒与0.25纳秒），能效提高了10 000倍（1000皮焦耳与0.1皮焦耳）。这些巨大的差异导致了缓存的出现，它们被用于减少访问内存带来的能耗和性能损失（参见第5章）。

2.3.2 常数或立即数操作数

程序经常会在一次操作中用到常数，例如，递增数组下标以指向数组的下一个元素。实际上，在运行 SPEC CPU2006 基准测试程序集时，超过一半的 RISC-V 算术指令有一个常数操作数。

只使用目前介绍过的指令，我们需要将常数从内存中取出才能使用。(这些常数会在程序加载的时候存放到内存中。) 例如，要将常数 4 加到寄存器 x22，可以使用以下代码：

```
ld  x9, AddrConstant4(x3)       // x9 = constant 4
add x22, x22, x9                // x22 = x22 + x9 (where x9 == 4)
```

假设 x3 + AddrConstant4 是常数 4 的内存地址。

避免使用加载指令的一种方法是提供另一个版本的算术指令，它的其中一个操作数是常数。这种带有一个常数操作数的快速加指令称为*立即数加*或 addi。要将 4 加到寄存器 x22，只需写成：

```
addi    x22, x22, 4     // x22 = x22 + 4
```

常数操作数经常出现；的确，addi 是大多数 RISC-V 程序中最常用的指令。通过把常数作为算术指令操作数，和从存储器取中出常数相比，操作速度更快，能耗更低。

常数 0 有另一个作用，通过有效使用它可以简化指令系统体系结构。例如，你可以使用常 0 寄存器求原数的相反数。因此，RISC-V 专用寄存器 x0 硬连线到常数 0。根据使用频率来确定要定义的常数，这是第 1 章中重要思想加速经常性事件的另一个实例。

自我检测 鉴于寄存器的重要性，芯片中的寄存器数量随时间变化的增长率是下面哪个？

1. 非常快：和摩尔定律一样快，摩尔定律预测每 18 个月芯片上的晶体管数量增长 1 倍。
2. 非常慢：由于程序通常以计算机语言实现，并且指令系统体系结构存在惯性，因此寄存器数量的增长速度与新指令系统在体系结构中的可行性保持一致。

详细阐述 尽管本书中的 RISC-V 寄存器为 64 位宽，但 RISC-V 架构师构思了 ISA 的多种变体。除了这种称为 RV64 的变体之外，还有一种具有 32 位寄存器的名为 RV32 的变体，成本降低的 RV32 能更好地适用于低成本的处理器。

详细阐述 RISC-V 中偏移量加基址寄存器的寻址方式对于数组和结构体来说非常适用，因为寄存器可以指向结构的起始地址，偏移量可以选择所需的元素。我们将在 2.13 节看到这样的例子。

详细阐述 数据传输指令中的寄存器最初是为了保存一个数组的下标，而偏移量用于指向数组的起始地址。因此，基址寄存器也被称为下标寄存器。而现在，内存容量大大增加，数据分配的软件模型更加复杂，所以数组的基地址通常在寄存器中传输，正如我们将看到的，它不再适合用偏移量保存。

详细阐述 从 32 位地址计算机到 64 位地址计算机的变化让编译器编写者不得不考虑 C 语言中数据类型的大小。显然，指针应该是 64 位，但整型应该是多少位？此外，C 语言有数据类型 int、long int 和 long long int。问题来自于将一种数据类型转换为另一种数据类型的时候，在 C 代码中出现了意外溢出，这种溢出并不完全符合标准，不幸的是该情

况并不罕见。下表显示了两个常见的选项：

操作系统	指针	int	long int	long long int
Microsoft Windows	64 位	32 位	32 位	64 位
Linux和大部分Unix	64 位	32 位	64 位	64 位

虽然每个编译器可能有不同的选择，但编译器通常会关联每个操作系统并做出相同的决定。为了让示例更简单，在本书中，我们将假设指针都是64位，并将声明所有的C整型为long long int以保持相同的大小。我们同样也遵循C99标准并声明用作数组下标的变量是size_t，这可以保证无论数组多大，它们的大小都是正确的。它们通常都被声明为long int。

2.4　有符号数与无符号数

首先来快速回顾一下计算机是如何表示数的。人们会习惯性地想到十进制，但其实数可以用任意基数表示。例如，十进制的123等于二进制的1111011。

数字以一系列高低电信号的形式保存在计算机硬件中，因此它们被认为是基数为2的数。(正如基数为10的数称为十进制数，基数2的数称为二进制数。)

由于在计算机中所有信息都由**二进制数位**（binary digit）或位（bit）表示，因此二进制数计算的“原子”单位是单个数位。有两种取值，可以被理解为：高或低，开或关，真或假，1或0。

二进制数位：也称作位。以2为基数表示，或0或1，作为信息的基本单位。

推广到任意基数，第i个数位d的值是

$$d \times 基^{i}$$

其中i从0开始并从右向左递增。显然，这种表示方法可以计算双字中的位所表示的数：简单地把该位用作基数的幂。下标10表示十进制数字，2表示二进制数字。例如，

1011_2

表示

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

在64位双字中，从右向左依次将位编号为0，1，2，3……下图展示了RISC-V双字内的位编号和数字1011_2的存放位置（为适合本书页面，将其分成两部分）：

63	62	61	60	59	58	57	56	55	54	53	52	51	50	49	48	47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
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	0	0	0	0
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	0	1	0	1	1

（64位宽，分成32位的两行）

由于双字既可以横向表示也可以纵向表示，因此最左侧和最右侧可能不明确。**最低有效位**（least significant bit）指的是最右边的位

最低有效位：以RISC-V为例，指双字中最右边的位。